VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Metodika a nástroj na zpracování rizik projektu

Methods and Tool for Process Validation

2018 Tomáš Mocek

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student:	**Bc. Tomáš Mocek**
Studijní program:	N2647 Informační a komunikační technologie
Studijní obor:	2612T025 Informatika a výpočetní technika
Téma:	Metodika a nástroj na zpracování rizik projektu Methods and Tool for Process Validation
Jazyk vypracování:	čeština

Zásady pro vypracování:

Cílem diplomové je zpracování metodiky a implementace nástroje pro evidenci a propojení rizik projektu s požadavky, splnění standardů. Budou vzaty v úvahu typy rizik a jejich provázání a zvolení vhodné metody pro daný typ rizika a jeho itegraci a sledování v projektu.

Práce bude obsahovat zejména:
1. Seznámení se s problematikou metodami řešení rizik.
2. Rešerše metodik a nástrojů a jejich provázání s jinými nástroji.
3. Volba architektury, design a implementace nástorje se zaměřením na very small enterprises.
4. Závěr, vyhodnocení.

Seznam doporučené odborné literatury:

[1] John F. Sowa, Knowledge Representation: Logical, Philosophical, and Computational Foundations, Brooks Cole Publishing Co., Pacific Grove, CA, ©2000
[2] Alec Sharp, Patrick McDermott: Workflow Modeling: Tools for Process Improvement and Application Development, Artech House; 2 edition (October 31, 2008)
[3] Pfleeger, Shari Lawrence, and Joanne M. Atlee. 2009. Software Engineering: Theory and Practice: Prentice Hall, ISBN 0136061699
[4] Pressman, Roger S. 2010. Software Engineering : A Practitioner's Approach. 7th ed. New York: McGraw-Hill Higher Education, ISBN 9780073375977
[5] Sommerville, Ian. 2010. Software Engineering. 9th ed, International Computer Science Series. Harlow: Addison-Wesley, ISBN 978-0137035151

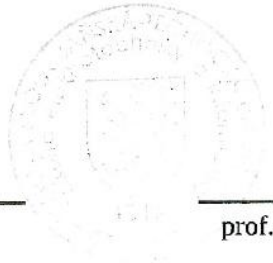
Další literatura podle pokynů vedoucího diplomové práce.

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Ing. Svatopluk Štolfa, Ph.D.**

Datum zadání: 01.09.2016

Datum odevzdání: 30.04.2018



doc. Ing. Jan Platoš, Ph.D.
vedoucí katedry

prof. Ing. Pavel Brandštetter, CSc.
děkan fakulty

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 1. dubna 2018 .....

Souhlasím se zveřejněním této diplomové práce dle požadavků čl. 26, odst. 9 Studijního a zkušebního řádu pro studium v magisterských programech VŠB-TU Ostrava.

V Ostravě 1. dubna 2018 .....

Zde bych rád poděkoval mému vedoucímu Ing. Svatopluku Štolfovi, Ph.D., za jeho odborné vedení a rady, díky kterým jsem byl schopen tuto práci dokončit. Dále bych rád poděkoval mým rodičům Ivě a Martinovi, za jejich nekonečnou podporu při studiu a poskytnutí výborných studijních podmínek, bez kterých bych se nikdy nedostal tam, kde jsem. V neposlední řadě bych chtěl poděkovat mému psovi Terezce, všem mým kamarádům, kteří se se mnou během studia kamarádili a také tvůrcům počítačové hry Heroes of Newerth, bez které bych tuto práci stihl dokončit o rok dříve.

Abstrakt

Tato práce se zabývá teoretickým popisem jednotlivých rizik softwarových projektů, se kterými se každé rozsáhlejší softwarové dílo musí potýkat, a dále pak implementací nástroje, který by měl tato rizika včas identifikovat. Primární funkcí nástroje je zobrazení a analýza metrik dostupných z databáze, popřípadě z integrací na externí systémy, jako je například nástroj Pivotal Tracker. Dále je možno specifikovat minima pro určité metriky a následně pozorovat, zda byla daná metrika splněna či nikoliv. Součástí aplikace je také možnost vytváření metrik nových, které lze pro potřeby uživatele přizpůsobit.

Klíčová slova: projektová rizika, projektové metriky, Pivotal Tracker, analýza rizik

Abstract

This thesis deals with theoretical description of software project risks, which each and every large scope project needs to deal with, as well as implementing software tool, which is supposed to identify them as soon as possible. Primary functionality of the tool is to display and analyze metrics available in the database, or use data from integrations to the external systems, for example Pivotal Tracker. It is also possible to specify minimal requirements for given metrics and then analyze, whether the metric passed these requirements or not. Another use case of the application is also creating custom metrics, which can be customized to meet a lot of user's needs.

Key Words: project risks, project metrics, Pivotal Tracker, risk analysis

Obsah

Seznam použitých zkratek a symbolů

QA	–	Quality Assurance
JSON	–	Javascript Object Notation
BSON	–	Binary Javascript Object Notation
NPM	–	Node Package Manager
SPA	–	Single Page Application
POM	–	Project Object Model
KISS	–	Keep It Simple, Stupid
DRY	–	Don't Repeat Yourself
UI	–	User Interface
API	–	Application Programming Interface
WAR	–	Web Application Resource
REST	–	REpresentational State Transfer
DAO	–	Data Access Object

Seznam obrázků

Seznam výpisů zdrojového kódu

1 Úvod

Management rizik softwarového projektu je klíčový pro vývoj software. Je to komplexní aktivita, která se zabývá tím, jak rešit nečekané události vznikající během vývoje a jejich příčinami.

Pro řízení rizik je v prvé řadě nutno identifikovat, jaká tato rizika jsou, a také co to vlastně riziko je. Článek vydaný v březnu roku 2011 v Hong Kongu[1] popisuje riziko jako *nejistý stav nebo událost, která pozitivně nebo negativně ovlivňuje cíle projektu*, popřípadě jako *nejistotu ve výsledku*.

V první kapitole této diplomové práce budou popsána základní rizika vývoje softwarového díla, která budou rozdělena mezi jednotlivé fáze vývojového životního cyklu. V druhé kapitole pak budou popsána rizika procesů softwarového vývoje, jako je například model *agile* nebo *vodopádový model* a v následující kapitole se bude práce věnovat popisu existujících nástrojů a metodik řešení rizik projektu.

Zbylá část dokumentu se pak věnuje popisu nástroje, který byl vyvíjen jako praktická součást této diplomové práce. V první kapitole, věnované praktické části, se práce zabývá obecnému popisu nástroje, k čemu slouží a jakou podporuje funkcionalitu. V následujících kapitolách pak bude popsán samotný návrh, vývoj a implementační detaily a problémy, které bylo nutno při tvorbě nástroje řešit.

V závěru pak bude průběh tvorby tohoto nástroje shrnut, budou navrženy další kroky a popsány vlastní myšlenky týkající nástrojů s podobným účelem.

2 Obecná rizika vývoje software

V této kapitole budou popsána obecná rizika každého většího softwarového projektu, se kterými se musí projektoví manažeři každodenně vypořádat. Informace budou členěny do tří podkapitol, kde první rozdělí rizika vývoje podle vývojová fáze od analýzy požadavků až po fázi údržby. Další podkapitola bude tyto rizika rozdělovat z pohledu procesů softwarového vývoje, jako je například proces vývoje vodopádovým modelem. V poslední podkapitole pak budou tato rizika podrobněji zkoumána a dále popsána.

2.1 Rizika softwarového projektu

Během celého životního cyklu vývoje software se setkáváme s různými riziky v různých fázích životního cyklu projektu, které mohou více či méně ovlivnit výsledný produkt. Tyto fáze můžeme rozdělit na Analýzu požadavků, fázi designu, implementace a testování a údržba software.

2.1.1 Analýza požadavků

Fáze analýza požadavků se skládá ze studie proveditelnosti, lákání požadavků ze zákazníka, jejich následná analýza a validace. Výsledkem pak je dokument požadavků.

V této fázi se nejčastěji setkáváme s organizačními problémy, jako jsou například neadekvátní estimace času, nákladů a zdrojů. Důsledkem naplnění těchto rizik projektoví manažeři často pracovně přetěžují programátory a dávají jim nereálné termíny uzávěrek, které jsou ve výsledku často nesplněny.

Dalším rizikem, zmiňovaným jako nejpravděpodobnější z rizik celého projektu, je nejasná definice a neúplnost požadavků zákazníka, které ale musí být jasné a srozumitelné pro analytiky i programátory. Bylo zjištěno, že zákazníci nejsou schopni popsat více než šedesát procent požadavků na začátku projektu[2]. Tyto požadavky se pak mění během celého životního cyklu a je nutno být na tyto změny připraven. Takové změny pak mohou být velmi časově i finančně nákladné a je nutné s těmito případnými výdaji navíc počítat a samozřejmě se je snažit omezit precizním plánováním a snahami o získání co nejvíce informací od zákazníka. V nejlepším případě pak od zákazníka můžeme získat i takové stěžejní informace, které ani oni sami neví.

2.1.2 Fáze návrhu

Během designu je nutno analyzovat dokument požadavků z předchozí fáze, vybrat správnou architekturu a programovací jazyk. Běžným problémem této fáze je, že dokument požadavků, který je výsledkem předchozí fáze, není jasný programátorům a architektům. K tomuto dochází v případě, že vývojáři nebyli zapojeni do analýzy požadavků. V takovém případě nelze začít tvořit design na základě přesných znalostí požadavků a mohou vyvinout design jiného systému, než o jaký je zájem. Dalšími riziky pak jsou nesprávná volba architektury, programovacího jazyka, popřípadě *overengineering* systému.

Overengineering je praktika, kdy se produkt designuje tak, aby byl více robustní nebo měl více funkcionalit, než je potřeba. Důvody pro tato rozhodnutí mohou být snahy o zajištění lepších než nutných funkcionálních limitů, připravení aplikace na případné budoucí rozšiřování, nebo omezení potencionálních nedostatků, které jsou pro drtivou většinu uživatelů považovány za přijatelné. Na první pohled výhodná praktika je obecně kritizována zejména kvůli navýšeným nákladům a potřebám na lidské a jiné zdroje, které často nebývají vyváženy výsledky, jelikož se systém nedostane do fáze, kdy by tato vylepšení byla potřebná. Další nevýhodou je to, že se celá aplikace stává složitější na porozumění a může ji tak být složitější dále udržovat.

Existují však oblasti, kde může být žádoucí. Jedná se o odvětví, kde bezpečnost nebo výkon určitého požadavků jsou stěžejní. Mezi tato odvětví patří například letectví, automotive nebo zdravotnictví[3].

Opakem overengineeringu je princip *KISS*, což je akronym pro *Keep it simple, stupid*, který říka, že většina systémů pracuje nejlépe, když jsou nadesignovány jednoduše a bez zbytečných komplikací[4].

2.1.3 Implementace a testování

Stejně jako v předchozí fázi je i zde riziko, že výsledek z předchozí fáze není srozumitelný, nejasný nebo je například příliš rozsáhlý, aby byli programátoři schopni začít pracovat.

Samozřejmostí jsou pak problémy na straně samotných programátorů, jako jsou syntaktické chyby, nevyužívání znovupoužitelnosti komponent a časté přepisování jednotlivých částí kódu. Princip, který cílí proti takovému opakování kódu se nazývá *DRY*, což je akronym pro *Don't repeat yourself*[5]. Velmi užitečnou technikou je také *code review*, kde jiná schopná osoba zkontroluje výsledný kód a odhalí případné nedostatky. Také se takto rozšíří znalost naprogramované funkcionality mezi více vývojářů.

Co se testování týče je velmi častým problémem nedostatečné testování a pokrytí kódu testy. Také je důležité opakující se činnosti automatizovat, čímž redukujeme prostor pro lidské chyby a velmi zrychlíme celý proces testování.

Pro zajímavost regresní testování celé základní funkcionality na jednom projektu, na kterém jsem se podílel, trvalo jednomu zkušenému testerovi okolo dvanácti hodin. Po zautomatizování tohoto procesu byl čas snížen na dvě hodiny, které vykonal sám testovací nástroj. Díky tomu se rapidně zefektivnily a zrychlily releasy a také se ušetřilo mnoho zdrojů, které bylo možno využít na jiných místech. Nevýhodou je, že tyto testy musí někdo vymyslet, naskriptovat a také je pak po každé závislé změně systému aktualizovat. Tyto potřeby jsou však zastíněny benefity, které automatické testování přináší.

2.1.4 Fáze údržby

Během údržby software se může stávat, že programátor nebude schopen reprodukovat problém nahlášený zákazníkem, nebo že bude kód těžce udržitelný kvůli volby špatné architektury, tech-

nologie, nebo nejasné a komplikované implementace existujících funkcionalit.

2.1.5 Společná rizika všech fází životního cyklu

Během celého životního cyklu projektu také existuje riziko, že bude omezován rozpočet a čas na plnění úkolů. V důsledku se pak může stát, že některé zásadní komponenty nebudou řádně dokončeny a otestovány. Samozřejmostí také je, že mohou zaměstnanci onemocnět, případně odejít za lepší prací.

2.2 Rizika procesů softwarového vývoje

V této podkapitole budou popsány základní populární software development frameworky jako jsou vodopádový model, inkrementální model nebo Agile development[6].

2.2.1 Vodopádový model

Vzhledem k povaze samotného modelu, který je neiterativní a z každé fáze se postupně přechází do fáze další, se reálným rizikům téměř nelze vyhnout. Zákazníci často neznají požadavky na začátku vývoje, což vede k potřebě redesignu, reprogrammingu a tím ke zvýšení nákladů i časové náročnosti.

Přestože je tento model často kritizován, tak jej více než třicet procent vývojářů stále využívá[7]. Další rizika tohoto vývojového procesu jsou obsažena v následujícím seznamu.

- Neustálé změny požadavků.
- Nepřekrývání jednotlivých fází.
  - Vzhledem k tomu, že každá fáze musí být úplně dokončena, než se přejde do fáze následující, část týmu odpovědná za jinou část musí často čekat na dokončení předchozí fáze, aby mohla jejich práce započít.
- Dlouhé vývojové fáze.
  - Dalším zdrojem rizik jsou relativně dlouhé fáze, které je těžké naplánovat a řádně ocenit. Také neexistuje žádný spustitelný produkt, dokud není vývoj téměř hotov. Zákazník tak nemá možnost produkt včas otestovat a odhalit případné nedostatky.

2.2.2 Inkrementální model

Inkrementální model je variantou vodopádového modelu, který se skládá z několika vodopádových cyklů, neboli iterací. Vznikl na základě rizik vycházejících z povahy vodopádového modelu popsaného v sekci 2.2.1. Každá iterace končí spustitelnou částí softwarového díla, která je předložena zákazníkovi a na základě jehož zhodnocení je naplánována a odstartována iterace následující. Tento proces je stále opakován, dokud není zákazníkovi dodán finální produkt.

I přes tyto snahy se však mohou vyskytnout různá rizika popsána v následujícím seznamu.

- Oddalování implementace některých požadavků
  - Developeři často odkládají implementaci požadavků, které jsou pak přidávány v pozdějších verzích, než by bylo žádoucí. V případě, že se jedná o stěžejní funkcionalitu, to může mít kritické následky v případě, že by v nich byly nalezeny chyby příliš pozdě a jejich oprava by se stala obtížnou.
  - Propagace chyb do pozdějších fází, kde jsou obtížněji napravovatelné.
- Nesprávné estimace času a rozpočtu
  - Dalším zdrojem rizik jsou relativně dlouhé fáze, které je složité korektně naplánovat a ocenit.

2.2.3 Agile

Agile je název pro několik iterativních a inkrementálních vývojových metodik, jako jsou například *Scrum* nebo *Extreme programming*. Základní principy těchto metodik jsou popsány v tzv. *Agile manifestu* a jedná se o prioritizaci individualismu před procesy, fungujícího software před rozsáhlou dokumentací, být schopen se adaptovat, místo držení se plánu a zvýšené kolaboraci se zákazníky.

Rizika těchto metodik jsou obsažena v následujícím seznamu.

- Pouze pro menší týmy
  - Agile metodiky nejsou vhodny pro rozsáhlé týmy, jelikož komunikace mezi nimi je složitější a bez zavedených procesů se již neobejde. V rozsáhlejších organizacích je nutno, respektive vhodno být schopen důležité akce identifikovat a mapovat, aby nevznikl chaos.
- Spoléhání se na jednotlivce
  - Agile se spoléhá na schopnosti a zkušenosti vývojového týmu namísto toho, aby byl řízen jinými lidmi. V případě, že je tým veden zkušeným team leaderem, tak je tato praktika velmi výhodná a v ideálním případě je tým, respektive individuální pracovník schopen autonomně vykonávat práci, který potřebuje pomoci pouze s prioritizací úkolů. V opačném případě se však z týmu stává zdroj rizik, která jsou schopna negativně ovlivnit celý projekt.
- Vzdálenosti mezi jednotlivými členy týmu
  - Je důležité, aby se jednotliví členové vývojového týmu mohli setkávat tváří v tvář, popřípadě aby alespoň probíhala každodenní komunikace. V dnešní době je velmi populární trend práce z domu, nicméně je třeba mít na paměti, že osobní komunikace je vždy ta nejefektivnější.

2.3 Podrobný popis jednotlivých rizik

Následující paragrafy popisují podrobněji jednotlivá rizika projektu rozdělena dle vývojových fází.

2.3.1 Fáze plánování a fáze získávání a vyhodnocování požadavků

- Nesprávné estimace projektového času, rozpočtu, rozsahu a jiných zdrojů
  - Projektoví manažeři mohou nesprávně odhadnout potřebný čas, cenu, rozsah a jiné zdroje potřebné k dokončení projektu. Toto vede k nereálnému projektovému plánu, rozpočtu, nejasném rozsahu a nedostatku zdrojů, což jsou problémy považovány za nejčastější příčinu selhání projektu.
- Nereálný časový plán
  - Plánovaný čas na celý projekt může přesáhnout původně dohodnuté datum dodání. V takových případech manažeři zahlcují developery prací, aby byl produkt dodán včas. Toto řešení je však často neúspěšné.
- Nereálný rozpočet
  - Naplánovaný rozpočet závisí na potřebném čase, usílí a zdrojích. V případě vyčerpání některého z těchto zdrojů dochází k selhání celého projektu.
- Nejasný rozsah projektu
  - Zvládnout odhadnout rozsah projektu (velikost, cíle a požadavky) je nejdůležitější úkol pro projektového manažera. Jejich odhadnutí je často náročné a chyba může způsobit, že budou zásadní funkcionality v projektu vynechány nebo jiné nedůležité brány v potaz. V obou případech lze očekávat neúspěšný výsledek projektu.
- Nedostatečné zdroje
  - Zdroje jako lidé, nástroje a technologie nemusí být dostatečné k dokončení projektu. V opačných případech systém nemusí být úspěšně dokončen při použití současných technologií a je potřeba využít technologie nové.
- Nejasné požadavky
  - Požadavky jsou nejasné, jsou-li nesrozumitelné pro vývojáře a analytiky.
- Neúplné požadavky
  - Požadavky jsou neúplné v případě, kdy neobsahují některou z uživatelských potřeb, restrikcí atd.

- Nepřesné požadavky
  - Požadavky jsou nepřesné, když nereflektují reálné uživatelské potřeby.
- Ignorování nefunkčních požadavků
  - Stává se, že se analytici a developeři zaměřují na to, co by měl systém dělat, a ignorují to, jaký by systém měl být (z hlediska použitelnosti, udržitelnosti, škálovatelnosti, schopnosti býti testovatelný atd.), přestože jsou nefunkční požadavky stejně důležité pro úspěšné splnění projektu.
- Konfliktní uživatelské požadavky
  - V případě více rozlišných uživatelů může docházet k tomu, že jsou jejich požadavky navzájem v konfliktu.
- Extra funkcionalita
  - Přidávání nadbytečné funkcionality za cílem systém vylepšit, může samotný projekt ohrozit.
- Neověřitelné požadavky
  - Požadavek je neověřitelný, jestliže nelze ověřit jeho splnění. Toto se stává v případě, kdy jej nelze otestovat, demonstrovat atd.
- Nekonzistentní požadavky
  - Požadavek je nekonzistentní, jestliže je v konfliktu s jiným požadavkem.
- Nedohledatelné požadavky
  - Pro účely dokumentace je nutné uvést zdroj požadavku pro případ budoucí dohledatelnosti.
- Nereálné požadavky
  - Požadavek je reálný v případě, že je jasný, ověřitelný, přesný, konzistentní, kompletní a uskutečnitelný.
- Nesprávně pochopené doménové termíny
  - Vývojáři používají jiné doménové termíny, které jsou neznámé pro většinou koncových uživatelů a může docházet k nedorozumění mezi jednotlivými stranami.

2.3.2 Fáze designu

- Dokument požadavků není developerům srozumitelný
  - V případě, že developeři nebyli přítomni při fázi analýzy a definice požadavků, může pro ně být dokument požadavků nesrozumitelný. V takovém případě nebudou schopni tvořit design na pevném základu znalostí a mohou vyvinout jiný systém, než je žádoucí. Také může dojít k tomu, že vznikne při případném ověřování požadavků časové zdržení.
- Nevhodná volba programovacího jazyku
  - Nevhodná volba programovacího jazyka může způsobit, že zvolený jazyk nebude podporovat navrženou architekturu, popřípadě omezí udržitelnost a přenositelnost. Je vhodné řídit se tvrzením *Use the right tool for the job*, což může být teoreticky jednodušší, než při jeho praktické aplikaci. Důvody jsou ty, že vybrat nejvhodnější technologii, může být z programovacího hlediska velmi důležité, avšak z hlediska úspěšného dokončení projektu za daných časových a finančních podmínek musí být také brány v potaz náklady na zaučení programátorů, na to, jak dobře se jiná technologie bude integrovat s již existujícími komponenty a podobně.
- Příliš komplexní systém
  - Komplexní a rozsáhlý systém může způsobit, že se developeři v kódu ztratí, nebudou vědět, kde začít pracovat a nebudou schopni systém rozložit do základních komponent. Stejný problém může nastat v případě příliš komplikovaného designu.
- Nedostatečně znovupoužitelné komponenty
  - V případě nekorektního odhadu ohledně znovupoužitelnosti může dojít k situaci, kdy developeři zjistí, že existuje mnohem více kódu, který je potřeba napsat celý znovu, místo aby byl použit již existující kód. V takovém případě pak dochází k nesplnění časových a rozpočtových estimací. Problém při navrhování komponent je, že ne vždy je jednoduché tyto znovupoužitelné komponenty předem identifikovat. Až časem je pak zjištěno, že by danou komponentu bylo vhodno zobecnit a znovupoužít, nebo naopak rozbít na několik samostatných komponent.
- Nevhodné umístění funkcí mezi komponenty
  - Nedokonalou dekompozicí systému může docházet k situacím, kdy developeři nejsou schopni jednoznačně rozhodnout, jaké komponentě přiřadit jednotlivé systémové funkce.

- Neúplné zdokumentování designu
  - Design dokument musí být dostatečně detailní, aby byli schopni developeři pracovat nezávisle.
- Nekonzistence design dokumentu
  - Nekonzistence většinou pochází z dulikace nebo překrývání mezi jednotlivými komponenty. Například jestliže více než jedna komponenta implementuje stejný funkční požadavek, budou v design dokumentu duplikace a redundance.

2.3.3 Fáze implementace a Testování

- Nečitelný design dokument
  - Jestliže je design dokument příliš rozsáhlý nebo nejasný, a tedy nesrozumitelný pro programátory, tak se může stát, že nebudou vědět, co programovat, nebudou schopni pracovat samostaně, popřípadě budou muset sami doplňovat mezery dokumentu.
- Moduly jsou vyvíjeny jinými programátory
  - V případě větších developerských týmů je nutné dodržovat stanovené programátorské standardy, nebo bude každá část vyvíjena jíným programátorským stylem a kód nebude dobře čitelný.
- Nedodržení standardů a best practices
  - V případě ignorování best practices může dojít k vytvoření komplexního, nekonzistetního a nejednoznačného kódu, který bude těžce udržitelný.
- Opakování kódu
  - Nedodržení principu DRY (Don't Repeat Yourself) způsobuje to, že je potřeba kódy opisovat, což stojí čas, snižuje udržitelnost a také dává prostor ke vzniku chyb.

    Ovšem nic není dobré brát ve světě programování jako dogma, a proto i zde je nutno nejdříve uvažovat a promyslet každý případ individuálně. Osobně se kloním k názoru, že není nutno za každou cenu a ve všech případech investovat do psaní znovupoužitelného kódu. Například při tvorbě UI komponent se mi zamlouvá taktika, kdy se kód začne psát pro potřeby znovupoužitelnosti až v případě, kdy je v aplikaci použit více, než dvakrát. Do té doby je to většinou zbytečná práce, která stojí čas.
- Nedostatečná automatizace testování
  - Přestože je testování opakující se proces, bývá často nedostatečně automatizován a vzniká prostor pro lidské chyby.

- Nedostatečný testovací proces
  - Testování bývá často považováno za dopňující proces vývoje, a ne jako jeho nezbytná součást. V takovém případě je mu věnována nedostatečná pozornost. Součástí tohoto rizika je například nedostatečná dokumentace jednotlivých testů popř. nedostatečné zaškolení v oblasti testování a může vyústit k neodhalení chyb, které se pak dostanou do ostré verze systému. Je nutné si uvědomit, že testování software je stejně důležité, jako jeho aplikace.
- Programování driverů a stubů
  - Při testování modulů jsou drivery a stuby součásti, které simulují jiné moduly potřebné pro dokončení testu. Programování těchto součástí bývá považováno za ztrátu času, jelikož nejsou součástí dodávaného díla.
- Nedostatečná regrese
  - Z časových důvodů se často stává, že regresní testování není zaměřeno na vše, co je potřeba a nemusí být odhalena kritická chyba.

2.3.4 Obecná rizika

- Neschopnost reprodukovat problém
  - V případě nalezených chyb je potřeba, aby byly dostatečně zdokumentovány tak, aby byly reprodukovatelné programátorem, který je má za úkol opravit. Po opravení chyby pak také musí být testerovi zřejmé, jak má takovou opravu otestovat.
- Rozpočet na údržbu systému
  - Údržba může být nejdelší vývojovou fází a její špatné odhadnutí může způsobit vyčerpání rozpočtu.
- Stále se měnící požadavky
  - Změny požadavků je nutno zahrnout do časového a finančního plánu, jinak dojde k jejich vyčerpání.
- Bus Factor
  - Bus faktor reprezentuje minimální počet členů týmu, kteří musí z projektu náhle zmizet (například být sraženi autobusem), aby se jeho vývoj zastavil z důvodu chybějících znalostí a kompetentního personálu[8]. Cílem projektového manažera by mělo být toto množství zvyšovat. Toho lze docílit například kvalitním dokumentováním a sdílením znalostí mezi co nejvíce členů.

3 Nástroje a standardy řízení rizik projektů a procesů

Následující podkapitola stručně popíše několik těchto nástrojů a přiřadí jim rizika z kapitoly 2. Další podkapitola se poté bude věnovat standardu vývoje software SPICE - ISO/IEC 15504.

3.1 Nástroje pro řízení projektů a jejich rizik

Na trhu existuje v současné době nepřeberné množství nástrojů pro řízení softwarových projektů a jejich rizik a je tedy možné si vybrat nástroj, který by přesně vyhovoval našim požadavkům. Většina osvědčených kvalitních nástrojů je pro privátní používání zpoplatněna, nebo jsou k dispozici pouze pro velmi malé projekty o málo členech.

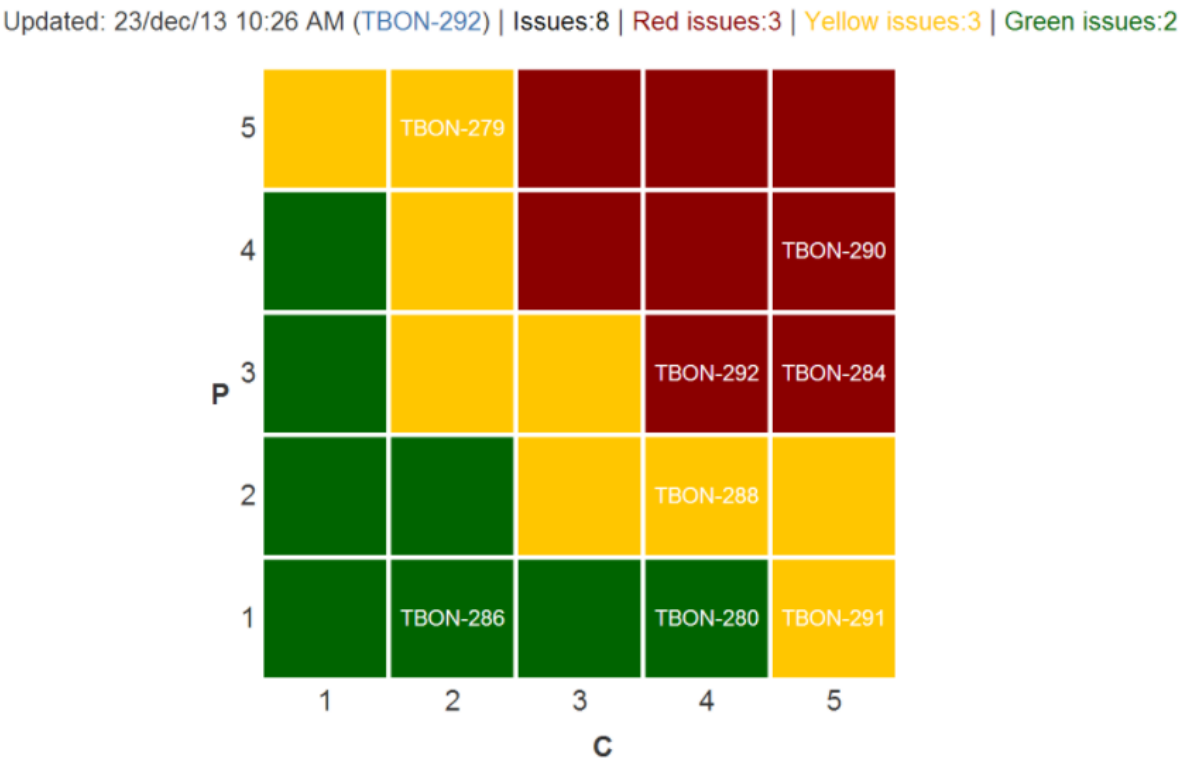
Obecně lze říci, že všechny tyto nástroje řeší vybraná rizika z kapitoly 2.3.1. Konkrétně se jedná o *nereálný časový plán*, *nejasné požadavky*, *neúplné požadavky*, *nepřesné požadavky*, *konfliktní uživatelské požadavky* nebo *nedohledatelné požadavky*. Další běznou funkcionalitou je zobrazení požadavků do různých typů grafů, jako jsou *Velocity chart* nebo *Sprint burndown*[9].

Všechna tato rizika samozřejmě vyžadují správnost zadání požadavku do systému a také to, aby uživatelé pravidelně aktualizovali čas, který vvývojem strávili a status, ve kterém se implementace požadavku aktuálně nachází.

3.1.1 Atlassian Jira + Agile

Jedná se o jeden z nejznámnějších, nejrozsáhlejších a nejpoužívanějších project management nástrojů, který se hodí do většiny projektů. Týmy si tento produkt mohou sami hostovat, nebo využívat cloudového řešení. Samozřejmě lze také integrovat s exisujícími nástroji Atlassianu, jako je například JIRA nebo Confluence. Projektoví manageři mohou navíc vytvářet customizovaná workflows nebo vizualizovat QA problémy. Výhodou je také mobilní aplikace a velké množství doplňků, kterými lze produkt doladit k přesným potřebám týmu. Jako nevýhoda se pak uvádí velká rozsáhlost nástroje, která zvyšuje čas potřebný k dokonalému ovládnutí nástroje[10].

Nástroj také podporuje plugin *Risk Management for Jira*[11], který umožňuje daná rizika dále sledovat. Plugin funguje na principu *matice rizik[12]*, kdy je každému riziku přiřazena hodnota určující pravděpodobnost výskytu incidentu a vážnost následků, které by tento incident způsobil. Obrázek 1 tuto matici rizik vyobrazuje a barevně rozlišuje požadavky dle kritičnosti.



Obrázek 1: Ukázka matice rizik v nástroji Jira[11]

Tato matice se pak může využít například pro eliminaci rizika *nedostatečná regrese*, kdy se při testování můžeme na tyto kritické požadavky více zaměřit.

3.1.2 VersionOne

Populární nástroj, který je jednoduchý na použití, přizpůsobitelný pro jakýkoli Agile styl (Scrum, XP a jiné), podporující reportovací funkcionalitu a jednoduchý drag and drop na Kanban board. Také ho lze synchronizovat s Jirou, GITem nebo Microsoft Visual Studiem. Jako nevýhodu je uváděno, že free verze je velice limitující.

Co se pokročilé funkcionality detekce rizik týče, podporuje nástroj velmi podobnou, ale omezenou funkcionalitu jako Jira. Také umožňuje každému požadavku přidělit hodnoty určující pravděpodobnost výskytu incidentu a vážnosti jeho následků, ale již není schopen tato rizika zobrazit do matice rizik[13].

3.1.3 Pivotal Tracker

Nástroj vyvinutý konzultační společností v oblasti vývoje software je určen pro mobilní a webové vývojáře. Podporuje více projektů, burndown grafy, komunikaci mezi uživateli, uživatelská stories a management projektových úkolů. Je relativně jednoduchý na naučení, má kvalitní iOS aplikaci a užitečnou množinu QA nástrojů. Je velmi vhodný pro Agile sofwarový vývoj a podpo-

ruje mnoho integrací jako například integraci s nástrojem Jira. Další výhodou, vzhledem k faktu, že většina podobných nástrojů je placených, je tento nástroj zdarma pro malé projekty (3 uživatelé + 2GB úložiště + 2 privátní projekty), veřejné zakázky, neziskové projekty a akademické instituce. Co se týče podpory pokročilejších metod pro podporu řízení rizik, tak tento nástroj bohužel další funkcionalitou nedisponuje.

Tento nástroj je také použit v implementovaném nástroji, kde se přes veřejné REST rozhraní získávají data přimo z Pivotal Trackeru.

3.1.4 ActiveCollab

Tento nástroj je popisován jako skvělé a dopstupné řešení pro malý business. Díky tomu, že je jednoduchý na používání, nemusí projektoví manageri učit, jak s tímto nástrojem zacházet. Podporuje emailovou komunikaci, prioritizování úkolů a funkcionality pro správu budgetů. Jako výhody jsou uváděny intuitivnost, kvalitní zákaznická podpora, iOS aplikace, time tracking a možnost omezení přistupu.

Stejně jako nástroj *Pivotal Tracker*, ani tento nástroj nepodporuje pokročilejší metody pro řízení rizik.

3.2 Standard SPICE - ISO/IEC 15504

ISO/IEC 15504 *Information technology - Process assessment*, také nazýván *Software Process Improvement and Capability Determination (SPICE)* je soubor technických standardů pro procesy softwarového vývoje a související business funkcionality[14]. Jedná se o referenční model, podle kterého lze posoudit schopnosti organizace dodávat produkty, jako je software, systémy a IT služby.

Tento referenční model se skládá ze dvou dimenzí. Jedná se o dimenzi procesů a dimenzi schopností.

3.2.1 Dimenze procesů

SPICE model definuje procesy rozdělené do pěti kategorií:

- zákazník - dodavatel,
- inženýrství,
- podpora,
- management,
- organizace.

3.2.2 Dimenze schopností

Pro každý proces je definována úroveň podle následujícího měřítka, kde jsou jednotlivé úrovně rozděleny do procesních atributů:

- Optimalizující proces - úroveň 5
  - inovace procesu,
  - optimalizace procesu.
- Předvídatelný proces - úroveň 4
  - měření procesu,
  - kontrola procesu.
- Stanovený proces - úroveň 3
  - definice procesu,
  - nasazení procesu.
- Řízený proces - úroveň 2
  - řízení výkonnosti,
  - řízení práce na produktu.
- Fungující proces - úroveň 1
  - výkonnost procesu.
- Neúplný proces - úroveň 0

Každý procesní atribut je dále posuzován podle následujícího měřítka:

- nedosažen (0 - 15%),
- částečně dosažen (>15 - 50%),
- velmi dosažen (>50 - 85%),
- úplně dosažen (>85 - 100%).

3.2.3 Přijetí SPICE modelu technickou veřejností

SPICE model je relativně populární a to například potvrzuje fakt, že má podporu v mezinárodní komunitě a bylo vykonáno více než 4000 zhodnocení dle tohoto modelu. Nejvyužívanější je zejména v oblastech, kde selhání některé části software může být kritické. Jedná se například o automotive nebo kosmický průmysl. Pro tato odvětví také existují derivace samotného modelu SPICE, jmenovitě *Automotive SPICE* a *SPICE 4 SPACE*.

4 Tvorba vlastního nástroje na zpracování rizik projektu

V současné době neexistuje mnoho moderních nástrojů, které by se věnovaly pokročilým technikám řízení rizik softwarového projektu, a proto je součástí této diplomové práce tvorba vlastního nástroje na zpracovávání rizik projektu.

Na rozdíl od nástrojů zmiňovaných v kapitole 3.1, se nejedná primárně o nástroj na správu požadavků, ale na analýzu metrik. Díky tomu by měl tento nástroj více dopomoci při analýze a zejména včasné identifikaci vznikajícího rizika, a to zejména pomocí uživatelsky vytvořených metrik a predikci požadavků, které budou popsány v podkapitole 4.2.

Tento metrikově orientovaný přístup také umožňuje nástroji, aby byla v tomto ohledu dále jednoduše rozvíjena.

4.1 Vize

Vize nástroje je, že bude sloužit jako systém, který je schopen schraňovat informace o důležitých projektových metrikách během vývoje softwarového díla. Tyto metriky je pak schopen zobrazit v přehledných grafech, které by měly být schopny projektového manažera a další uživatele včas upozornit na nebezpečné trendy, které by mohly vést k ohrožení úspěšnosti projektu. Všechny metriky jsou vázány na jednotlivé sprinty, a proto lze tento nástroj použít pouze na projekty, které používají iterativní vývojové cykly. Jiné než iterativní metodiky nejsou podporovány z toho důvodu, že jsou velmi nevhodné na jakékoli aplikace většího rozsahu a výber neiterativní metodiky tak představuje samo o sobě výrazné riziko.

4.2 Základní funkcionalita

Podporované metriky lze rozdělit do dvou kategorií.

1. Bežné metriky

   Mezi tyto metriky patří například vyhodnocení poměru počtu naplánovaných požadavků za určitý sprint k počtu splněných požadavků. Mezi další metriky pak patří porovnání naplánovaného rozpočtu ke skutečným výdajům, nebo zobrazení informací o tom, kolik defektů bylo nalezeno během vývoje, po releasu popřípadě kolik jich bylo nalezeno zákazníkem.

   Tyto metriky nejsou v této fázi vývoje svázány s požadavky zákazníka, ale je možné je o tuto funkcionalitu jednoduše rozšířit a může se tedy jednat o návrh pro budoucí rozšiřování aplikace. V případě nouze o tyto požadavky je pak možné využít uživatelsky nadefinovaných metrik, které je možné přizpůsobit jakýmkoli potřebám.

2. Zákaznické metriky

   Jedná se o požadavky, které jsou svým významem zajímavější pro zákazníka. Zákazníka může například zajímat, kolik je jeho business požadavků namapováno na požadavky vý-

vojářské. Další takovou sledovanou metrikou je, zda byly jeho požadavky namapovány na testy, nebo jaké procento softwarových jednotek bylo pozitivně, respektive negativně ověřeno.

U těchto metrik lze také stanovit, jaký procentuální cíl byl zákazníkem stanoven. Například může zákazník požadovat, aby byla funkcionalita osmdesáti procent požadavků ověřena testy. V případě, že tento požadavek není splněn, uživatel je varován tím, že mu budou informace zobrazovány v červených číslech.

Graf takové metriky pak také zobrazuje minimální průměrné hodnoty, kterých je potřeba dosáhnout, aby byly tyto zákaznické požadavky splněny. Graf tady navíc zobrazuje nejhorší možnou datovou sadu, která by stále splňovala stanovené minimum.

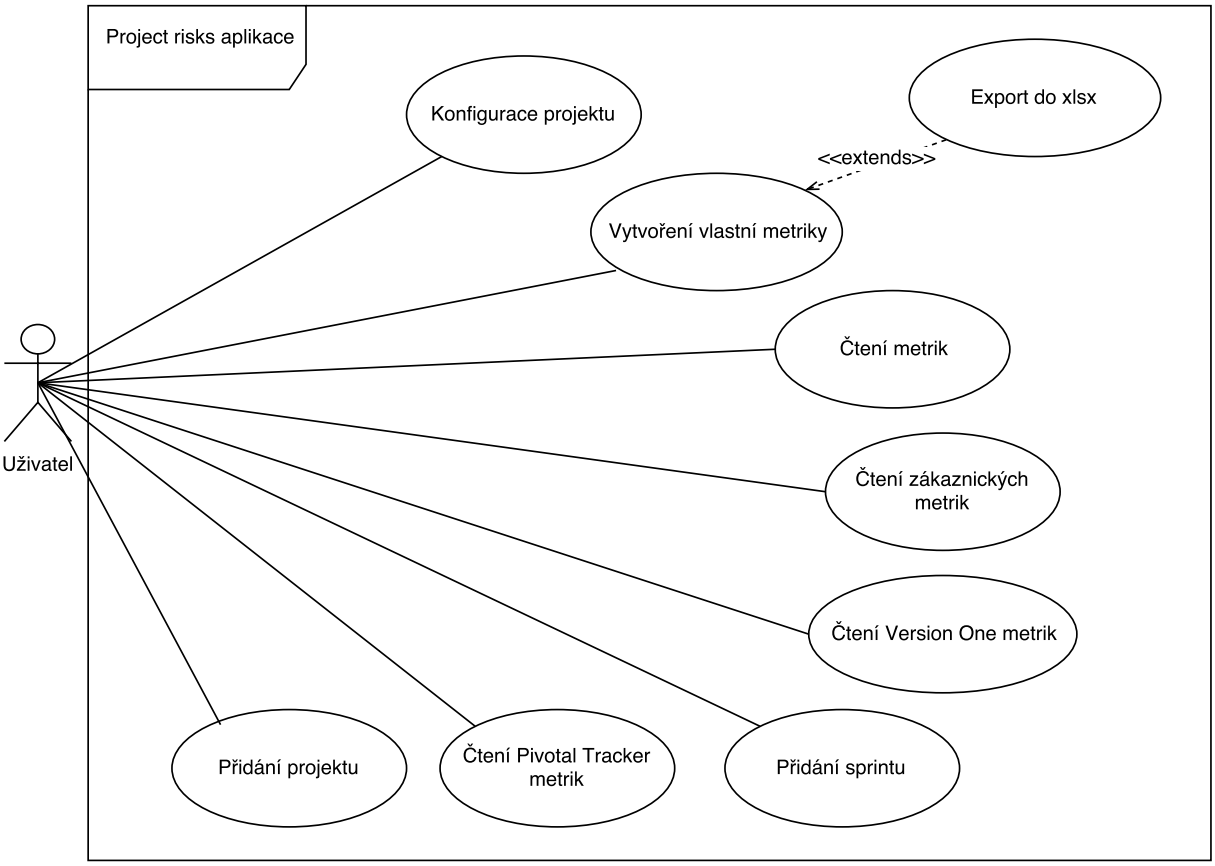
Při využití projektové konfigurace *Expected Number of Sprints*, tedy předpokládaný počet sprintů v projektu, jsou grafy doplněny o *budoucí* sprinty, které ještě nemají skutečné hodnoty, ale jsou dopočteny z již známých dat, zprůměrovány a znázorňují taková čísla, kterých je nutno dosáhnout, aby byly na konci projektu tyto metriky splněny. Implementační detaily této funkcionality popisuje kapitola 7.6.

Další důležitou funkcionalitou je integrace na existující *issue tracking* nástroje, jako je *Pivotal Tracker* nebo *Version One*. Cílem zde je, aby bylo co nejvíce informací, které rozhraní integrovaných nástrojů podporuje, namapováno na existující metriky vyvíjené aplikace. V tuto chvíli existuje pouze mapování na metriku *Velocity*, která pro daný sprint měří množství dokončených požadavků v jejich celkovém množství. Rozšířit aplikaci o další metriky je jeden z návrhů na její vylepšení, avšak toto záleží zejména na rozhraní daných nástrojů.

Posledním a také nejnáročnějším use case na vývoj jsou uživatelsky vytvořené metriky. Tato funkcionalita podporuje vložení na graf a porovnání jakýchkoli typů dat v databázi vůči jakémukoli jinému. Uživatelsky vytvořené metriky podporují zobrazení dvou datových množin. Obě tyto množiny musí být v databázi. Například je tedy možné na graf nanést množství nalezených bugů k počtu splněných požadavků podle určitého výrazu. Tento výraz pak může například nabývat hodnot výše zmíněných datových množin v podílu.

Důležitou součástí tohoho use case je specifikování cíle, který by měla tato vytvořená metrika splňovat. Předpokládejme, že byla vytvořena metrika, která vyjadřuje podíl mezi skutečným rozpočtem a počtem splněných úkolů. Nyní můžeme specifikovat, že náš cíl je, aby tento podíl byl menší než jeden tisíc. Do grafu se pak v tomto případě zanesou jak skutečné hodnoty, tak i hodnody takové, aby byl tento cíl v minimální možné míře splněn. Data z těchto metrik lze také vyexportovat do Excelu. Více podrobností včetně implementačních detailů je také možno najít v kapitole 7.5.

Všechny možné případy užití aplikace znázorňuje obrázek 2.



Obrázek 2: Use case diagram aplikace

Z obrázku 2 lze také vyčíst, že v tuto chvíli nástroj nepodporuje žádné přístupové role. Jakýkoli uživatel tedy může vykonávat jakékoli operace. Pokud by požadavek na omezení přístupových práv někdy nastal, je možno aplikaci o tuto funkcionalitu jednoduše rozšířit. Diagram dále zobrazuje use case, který nelze přímo z aplikace vykonávat. Jedná se o use case *Add new sprint*. V tuto chvíli je tato funkcionalita dostupná pouze přes backend Java rozhraní, které popisuje kapitola 5.2.7 a je nadefinované ve třídě *ProjectController.java*. Pro použití tohoto use case je tedy nutné použít nástroje jako je *Postman* nebo *Soap UI*.

Aplikace byla také nasazena v produkční verzi na vlastní server na adrese

http://37.46.85.193:8090/metrics#Main.

5 Architektura a návrh nástroje

Cílem této kapitoly je seznámit s implementační částí samotného nástroje, s jeho architekturou, návrhem a použitými technologiemi.

V první podkapitole se bude práce zaměřovat na velmi stručný popis nejdůležitějších technologií, které byly v práci použity a v následujících podkapitolách pak bude popsána architektura systému. Seznam nejdůležitějších technologií pak bude dále doplněn o více detailní popis jednotlivých technologií a také doplněn o méně důležité technologie pro backend, respektive frontend.

Vzhledem k tomu, že se aplikace skládá z frontend části a backend části, je tato kapitola rozdělena tak, aby byl zřejmý přechod mezi těmito vrstvami a je také doplněna o grafy a schémata tak, aby pomohly porozumět dané problematice.

5.1 Použité technologie

Následující technologie jsou základním stavebním kamenem celé vyvíjené aplikace a jejich základní znalost je nutná k pochopení navržené architektury.

5.1.1 Java 8

Aplikace používá syntaxi Javy 8 jako je *Optional* nebo *Stream API*, kdekoli je to možné. Krátce po release Javy 9 byly vynaloženy snahy o její použití, nicméně tyto snahy ztroskotaly na nedostatečné podpoře Javy 9 ve Spring Bootu. Přestože je Java 9 velmi zajímavým updatem, tak by vzhledem k faktu, že je většina aplikace napsána v Javascriptu, upgrade neměl značné opodstatnění a největší výhodou by bylo zejména naučení se upgradovat Java verzi na Spring Boot aplikacích. Jedna ze zajímavostí, která by se dala využít i z praktického hlediska, by pak bylo použití nového HTTP/2 API na HTTP requesty. Co se funkcionality týče by ovšem díky upgrade nic nezměnilo.

V každém případě je vždy dobré udržovat verzi Javy aktuální a proto je tento upgrade jeden z možných témat pro budoucí vývoj.

5.1.2 Spring Boot

Jedná se o variantu klasického Springu, která se řídí konvencí *convention-over-configuration*[15], což je paradigma softwarvých frameworků založené na omezení počtu rozhodnutí, které musí developer udělat, aniž by přicházel o jejich flexibilitu. Toto ve výsledku znamená, že ve značné míře není třeba aplikaci konfigurovat, jelikož je již podle konvencí nakonfigurována *out of the box*. V případě, kdy by bylo potřeba nastavení aplikace změnit či přidat vlastní, nejjednodušší cestou je soubor *application.properties* v adresáři *resources*.

V případě této aplikace bylo mimo jiné potřeba nakonfigurovat jméno MongoDB databáze pomocí konfigurace *spring.data.mongodb.database = project-risks* a také bylo změněno číslo portu z hodnoty 8080 na hodnotu 8090.

Spring Boot je dále schopen tvořit samostatné Spring aplikace, umí využít Jetty nebo Tomcat server bez potřeby deploye WAR souborů, zahrnuje startovací POM soubory pro usnadnění práce s Mavenem a automaticky konfiguruje Spring, kdekoli to je možné[16].

5.1.3 ReactJS

React, někdy nazýván také React.js nebo ReactJS je open-source Javascriptová knihovna pro tvorbu uživatelských rozhraní. Je udržována Facebookem, Instagramem a komunitou dobrovolných developerů. Základním stavebním kamenem aplikace napsané v Reactu jsou komponenty, které cílí na znovupoužitelnost[17].

5.1.4 Redux

Stavový kontejner pro JavaScriptové aplikace, který pomáhá psát aplikace, které se chovají konzistentně, běží ve vícero prostředích a a jsou jednoduché na testování[18]. Zjednodušeně by se dalo říci, že se Redux stará o uchování a správu dat v celé aplikaci.

5.1.5 MongoDB

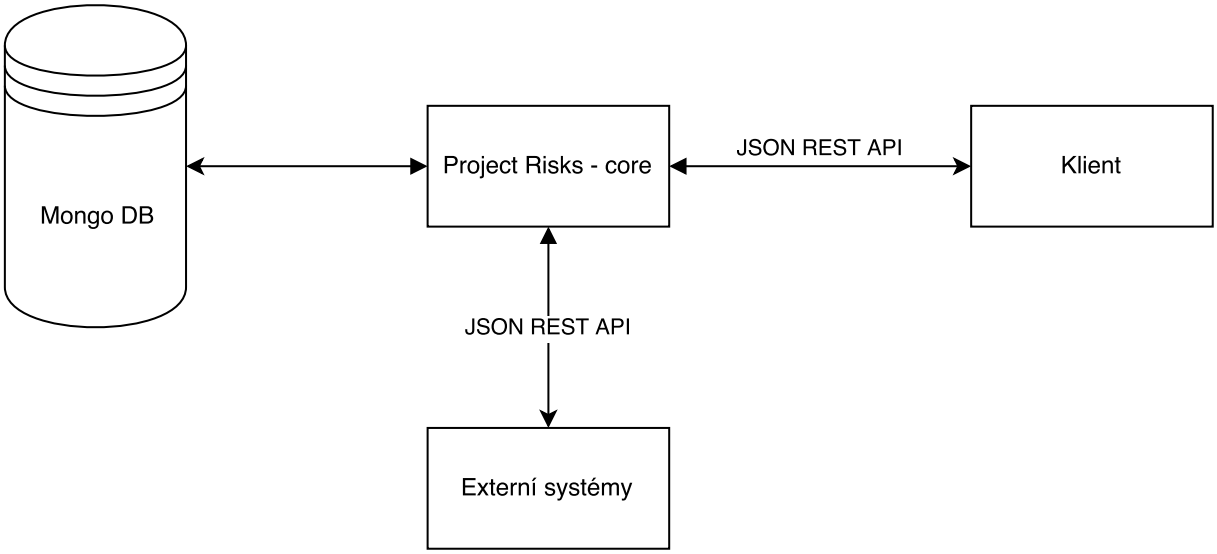
Volně dostupná open-source multi platformní dokumentovaně orientovaná databázová platforma. Je klasifikovaná jako NoSQL[19] databáze a data ukládá ve formě BSON, což je forma JSONu[20]. MongoDB nemá pevně danou strukturu dat, které lze do databáze uložit a také neukládá data do tabulek, ale do objektů, které se nazývají kolekce. Záznamy těchto kolekcí pak jsou dokumenty, které mohou být velmi složité struktury a mohou obsahovat další vnořené objekty nebo pole.

5.1.6 Konfigurační management a testování

Co se konfiguračního managementu týče byly použity klasické technologie jako jsou GIT a Maven. Pro testování pak byl použit framework JUNIT verze 4 a mockovací framework Mockito. Pro potřeby logování aplikace využívá nástroje Log4J2.

5.2 Architektura a návrh

Aplikace se skládá ze dvou základních částí, a to z backendu napsaného v jazyce Java a frontendu napsaného v Javascriptu. Obrázek číslo 3 ilustruje high-level pohled na kompletní aplikaci.



Obrázek 3: High-level architektura systému

5.2.1 Architektura a návrh backend části

Backend je členěn do dvou základních Maven artefaktů. Prvním z těchto artefaktů je *projectrisks-api*. Druhý, který tvoří jádro celého systému je *projectrisks-core*.

Obsahem artefaktu *projectrisks-api* není žádná business logika. Jediné, co obsahuje, jsou definice DAO tříd, které se používají v endpointech artefaktu *projectrisks-core*, který má tento artefakt zadefinován jako závislost. Na tyto objekty se pak mapují těla požadavků z, respektive na externí endpointy, jako je například API nástroje Pivotal Tracker.

Důvod pro toto rozdělení není v této fázi vývoje aplikace zřejmý, nicméně je to vhodné pro případ, kdyby vznikla aplikace, která by se na vytvářenou aplikaci chtěla integrovat. V tom případě by přidáním tohoto artefaktu do svého POM vznikla okamžitá znalost dat, které API aplikace podporuje.

Druhý jmenovaný artefakt, *projectrisks-core* je hlavní artefakt backend části projektu. Základní funkcionalitou tohoto artefaktu je, že slouží jako middleware pro komunikaci mezi frontendovou částí a extérními systémy, jako je například napojední na API nástroje Pivotal Tracker, popřípadě mezi databází.

Databáze je zde použita NoSQL MongoDB popsaná v kapitole 5.1.5. Motivací pro volbu tohoto druhu databáze byly zejména osobní cíle a sympatie, jelikož jsem se s tímto druhem databáze setkal pri implementaci semestrálního projektu, který byl implementován v .NET fra-

meworku. Jelikož jsou API pro .NET i Javu poměrně odlišné, tak jsem se chtěl naučit pracovat i s Javovskou verzí.

Jádro celé aplikace tvoří Spring Boot popsaný v kapitole 5.1.2 z maven parent artefaktu *spring-boot-starter-parent*, díky kterému se lze vyhnout některým Maven konfiguracím závislostí a pluginů.

Samotná aplikace je členěna do několika vrstev, které byly implementačně rozděleny pomocí balíčků.

5.2.2 Vrstva pro práci s databází

Nejnižší vrstvou je vrstva pro správu a přístup do databáze umístěná v balíčku *repository*. Existuje zde třída *MongoDBConnector*, což je singleton třída, která se stará a připojení na databázi a také vrací spojení pro operace s daty. Další třídy obsahují operace pro ukládání dat do databáze, popřípadě jejich získávání.

Pro práci s Mongo DB databází byl kromě MongoDB závislosti použit externí balíček *Morphium*[21], který se stará o objektové mapování na databázové struktury, což je funkcionalita, která bohužel není součástí oficiálního Mongo DB balíku. Velkou nevýhodou je velmi neaktuální a nečitelně psaná dokumentace, která velmi rapidně snižuje schopnost začít rychle využívat tyto funkcionality. Z tohoto důvodu bylo občas potřeba zkoumat přímo zdrojové kódy balíku, aby bylo možno vyřešit některé problémy.

Během implementace nástroje jsem se bohužel setkal s neřešitelným problémem týkající se této knihovny. Hlavním problémem bylo, že odkaz s dokumentací se stal neaktivním a také jsem objevil bug, popřípadě chybějící feature, kde knihovna nebyla schopna namapovat vnořená pole objektů.

Díky tomuto problému byla nakonec použita oficiální knihovna *Spring Data MongoDB*[22], což se však ukázalo jako velmi dobrá varianta, protože je tato knihovna oficiálně vyvíjena a udržována, je velmi dobře integrovatelná do Spring Boot a celkově velmi usnadnuje práci díky tomu, že její třídy dědí ze *Spring Data Commons* projektu[23]. Hlavní třídou této verze implementace je pak rozhraní *ProjectRepositorySpring*, jehož funkční popis bude dále popsán v kapitole 7.1, která se týká implementačních detailů.

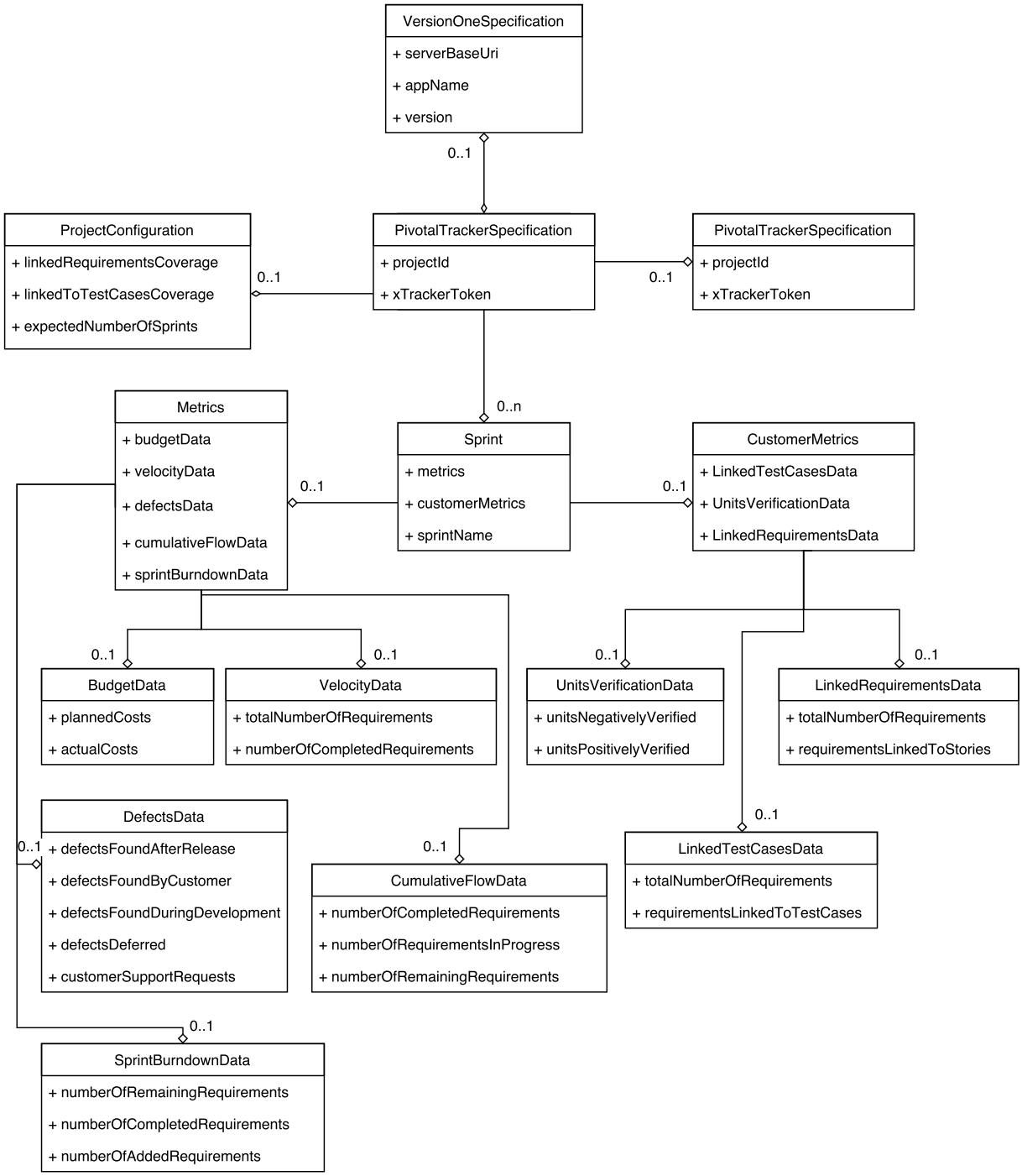
5.2.3 Návrh databáze

Vzhledem k tomu, že aplikace využívá NoSQL databáze, je její návrh velmi odlišný od návrhu klasické databáze SQL. Celá databáze se skládá z jednoho databázového objektu, tedy kolekce. Kolekce v MongoDB slouží k uchovávání dokumentů, tedy záznamů, a s nadsázkou by se dala přirovnat k tabulce v klasické SQL databázi. Aby vznikly potřebné kolekce, stačí vybranou třídu oanotovat anotací *@Document*. V případě této aplikace se jedná o *@Document(collection = "projects")*.

Každý záznam (dokument) této kolekce reprezentuje jeden z projektů, který tato aplikace spravuje a obsahuje název projektu, který musí být unikátní. Dále pak obsahuje jednotlivé vývojové sprinty, které pak obsahují data například o tom, kolik požadavků je na tento sprint naplánováno, jaký je rozpočet apod. Důležitá informace o každém projektu je pak také konfigurace toho, jaká jsou minima pro splnění určitých metrik. Pod touhle informací si lze například představit to, že si projektový manager, respektive zákazník, může nadefinovat, že požaduje, aby měl projekt určité procento všech požadavků namapovaných na testovací use case.

Posledním typem dat je pak informace o integraci na externí nástroje, jako jsou *Pivotal Tracker* nebo *Version One*.

Detailněji je struktura datového modelu popsána na obrázku číslo 4.



Obrázek 4: Schéma datového modelu

5.2.4 Servisní vrstva

Třídy servisní vrstvy umístěné v balíčku *services* se starají o business logiku a také volají metody z jiných vrstev. Příkladem takové servisní operace může být volání metody z vrstvy pro práci s databází viz kapitola 5.2.2, nebo volání metody z vrstvy pro přístup k externím zdrojům viz kapitola 5.2.5. Také se zde využívá tak zvaná metoda *Coding to Interfaces*[25], kdy ke každé třídě existuje její rozhraní, které je pak pomocí Spring anotace *Autowired* napojeno na potřebné implementační třídy.

5.2.5 Vrstva pro přístup k datům z externích zdrojů - connector vrstva

Vrstva, která přistupuje na REST rozhraní jiných systémů, jejichž implementace nebyla součástí tohoto nástroje. Jeden z příkladů takového externího systému je nástroj *Pivotal Tracker* viz kapitola 3.1.3. Její jediná povinnost je pomocí Springové třídy *RestTemplate* navázat spojení na zadanou externí URL, podle HTTP metody vrátit, respektive zaslat data a tato data dodat servisní vrstvě, která toto volání vyvolalo. Tato data se poté mapují pomocí *Jackson Databind* data-binding projektu na vlastní objekty nástroje[26]. Connector vrstva také využívá techniky *Coding to Interfaces* zmíněné v kapitole 5.2.4.

Konkrétně se aplikace integruje na API od nástroje *Pivotal Tracker* a *Version One*. Tato integrace bude dále popsána v kapitole 7.2.

5.2.6 Vrstva mapperů

Tato vrstva slouží pro mapování dat z RESTových rozhraní na interní objekty aplikace a naopak.

5.2.7 Vrstva endpointů

Vrstva sloužící pro odchytávání požadavků na aplikaci. Definuje endpointy, tedy rozhraní, na které se lze napojit a lze tak komunikovat s aplikací. Nejčastější použití pro tento use case je, že se frontend napojí na požadovaný endpoint, získá přes něj data a zobrazí je ve svém view. Po namapování požadavku na konkrétní endpoint dochází k volání servisní metody, která tento požadavek zpracuje.

Mezi tyto endpointy patří následující operace:

1. **GET**: */rest/internal/v1/projects* - získání všech projektů
2. **POST**: */rest/external/v1/project* - vytvoření nového projektu

   Při duplicitním jménu projektu vrací endpoint HTTP status kód 409.
3. **POST**: */rest/external/v1/project/:projectName/sprint* - přidání nového sprintu k danému projektu

   Při duplicitním názvu sprintu vrací endpoint HTTP status kód 409. Při neexistujícím názvu projektu vrací endpoint HTTP status kód 404.

4. **POST**: */rest/external/v1/project/:projectName/config* - vytvoření konfigurace pro daný projekt

   Při neexistujícím názvu projektu vrací endpoint HTTP status kód 404.

5. **GET**: */rest/external/v1/project/:projectName/pivotal/iterations* - získání všech Pivotal Tracker iterací pro daný projekt

6. **GET**: */rest/external/v1/project/:projectName/version_one* - získání všech Version One sprintů pro daný projekt

5.2.8 Doménové třídy

Mimo zmíněné vrstvy existují také třídy, které nám pomáhají s chodem aplikace. Základním stavebním kamenem samotné aplikace jsou samozřejmě doménové třídy uchovávající data. Tyto třídy jsou umístěny v balíčku *domain*. Na tyto třídy jsou také mapovány data z externích zdrojů.

5.2.9 Validační balíček

Tento balíček obsahuje třídy pro validaci dat, a to konkrétně validaci dat z příchozích požadavků. Díky těmto třídám můžeme elegantně ověřit, že vstupní data do aplikace nenabývají hodnoty null, že určitý parametr je korektního datového typu a podobně. Při neúspěšné validaci vyhazuje vlastní nadefinovanou výjimku *RestRequestValidationException* dědící ze třídy *Exception*.

Hlavní třídou validace REST požadavků je třída *RestRequestValidator*, která implementuje návrhový vzor *Builder* a díky kterému můžeme validaci elegantně řetězit.

5.2.10 Ostatní použité technologie v aplikaci

Mimo již zmíněné externí balíčky byly také použity následující zavislosti, které se přímo nepodílejí na běhu aplikace, ale jsou velmi důležité pro vývoj a hledání případných chyb.

5.2.11 JUNIT

Pro potřeby testování byl použit typický JUNIT framework. Vzhledem k faktu, že backend neobsahuje složitou business logiku, tak je testy pokryta zejména vrstva mapperů. Pro tyto potřeby byla také vytvořena tovární třída obsahující metody na generování testovacích dat *MapperDataFactory.java*, díky které není nutno vytvářet testovací objekty v testech samotných a ty se tak stávají čistějšími a čitelnějšími. Některé testy, zejména ty vyšší úrovně, používají také testovací framework *Mockito*.

Takový test znázorňuje následující výpis zdrojového kódu.

```
1  @Test
2  public void mapProjectDTO() {
3      ProjectExternalDTO projectDTO = createProjectExternalDTO();
4
5      ProjectMapperExternal mapperSpy = spy(projectMapperExternal);
6      Project project = mapperSpy.mapProjectDTO(projectDTO);
7
8      verify(mapperSpy, times(1)).mapPivotalTrackerSpecification(any(), any());
9      verify(mapperSpy, times(1)).mapVersionOneSpecification(any(), any());
10
11     assertEquals(PROJECT_NAME, project.getProjectName());
12 }
```

Výpis 1: mapProjectDTO test

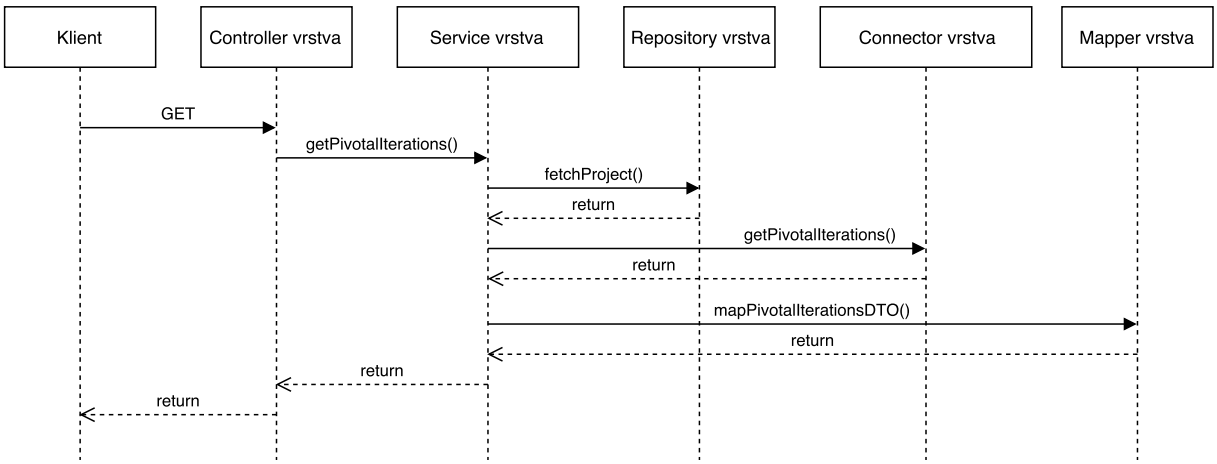
Ukázkový test na řádku číslo 3 volá právě metodu na vygenerování testovacích dat, takže je tento test, popřípadě tato třída od tohoto kódu očištěna. Také zde vidíme použití frameworku *Mockito*, kde ověřujeme, že zavoláním metody *mapProjectDTO()* byly dále vyvolány metody *mapPivotalTrackerSpecification()* a *mapVersionOneSpecification()*. Samotné otestování těchto volaných metod je již zodpovědnost jiných testů.

5.2.12 Log4J2

Logování je naprostou nutností moderních aplikací. Pro potřeby této aplikace byl použit moderní logovací nástroj Log4j 2[27] a jeho konfigurace je umístěna v XML dokumentu *log4j2.xml* v adresáři *resources*. Aplikace využívá tří nakonfigurovaných loggerů, kde dva logují do souborů a jeden do konzole aplikace.

Do konzole se logují veškerá hlášení s úrovní vyšší než úroveň *TRACE* a slouží zejména pro učely vývoje. Co se souborových loggerů týče, tak existuje logger *MongoAppender*, který loguje do souboru *mongo.log* veškeré operace týkající se databáze s log úrovní vyšší než *DEBUG* a logger *ErrorAppender*, který loguje do souboru *error.log* veškeré zprávy s úrovní vyšší než *WARN*.

5.2.13 Ukázkové použití vrstev



Obrázek 5: Příklad komunikace mezi vrstvami při volání dat z Pivotal Trackeru

Obrázek číslo 5 popisuje, jak prochází jeden požadavek celou aplikací. Jak lze vidět, tak request odchytí vrstva controllerů, která ji předá servisní vrstvě. Tato vrstva se pak stará o řízení veškeré logiky, která je potřeba pro vykonání požadavku udělat. V tomto případě se jedná o získání informací o projektu z repository vrstvy, respektive z databáze. Jakmile servisní vrstva tato data obdrží, může iniciovat volání connector vrstvy, která vyřídí samotnou komunikaci mezi aplikací a Pivotal Tracker serverem a vrátí data. Tyto data pošle opět servisní vrstva do vrstvy mapperů, aby byla získaná data namapována do doménových objektů. Výsledek z vrstvy mapperů je pak již jen postupně vracet do servisní vrstvy, poté do controller vrstvy a nakonec data doputují zpět ke klientovi, který si je vyžádal.

5.3 Architektura frontend části

Frontend je napsán kompletně v Javascriptu a jeho odpovědností je zobrazovat data uživatelům a zpřístupňovat jim UI rozhraní pro práci s aplikací. Základní použitá Javascript knihovna je ReactJS popsaná v kapitole 5.1.3. Jako nástavbu pro ReactJS byl použit stavový kontejner Redux[18], který nám umožňuje centralizovaně uchovávat stav (data) celé aplikace v jednom objektu, a také nám zpřístupňuje rozhraní pro jeho manipulaci. Verze Javascriptu je aktuálně 6th Edition - ECMAScript 2015[24], což znamená, že nelze využít některou funkcionalitu z novějších verzí, jako je třeba *spread operator* z ECMAScript 2016 nebo *await/async* z ECMAScript 2017. Tyto vlastnosti novějších verzí nedodávají žádnou novou funkcionalitu, jelikož pouze dodávají nové, příjemnější možnosti, jak Javascriptový kód psát. Programátorům však značně ulehčují život, a proto by bylo v budoucnu velmi vhodné, aby k navýšení verze došlo.

Jedná se o *Single Page Application*[28], což znamená, že celá aplikace se skládá z jedné hlavní HTML stránky, která dynamicky mění se její obsah. Nikdy tedy nedochází ke kompletnímu znovunačtení stránky, což zrychluje celou aplikaci a také působí velmi příjemným dojmem pro uživatele.

Popis frontend části z hlediska návrhu uživatelského rozhraní je pak k dispozici v příloze této práce.

5.3.1 Další použité technologie

1. Node.js

   Přestože je Node.js zejména Javascriptové backend prostředí a tato aplikace používá na backendu Javu, tak je i zde použit Node.js. Konkrétně se používá jeho package manager *npm*[29][30].

2. Webpack

   Open-source module bundler pro Javascript, který je schopen z mnoha zdrojových souborů a jejich závislostí vytvořit jeden cílový soubor, neboli bundle. Díky této techniky je potřeba, aby prohlížeč stáhl pouze jeden soubor, což zlepšuje výkon [31].

3. Babel

   Překladač, který převádí Javascriptový kód z vyšších verzí do verze nižší. Tato technika je potřeba z toho důvodu, že internetové prohlížeče doposud nebyly schopny podporovat nové funkcionality a stránka by teda byla nefunkční[32]. Tento problém je nejvíce patrný u prohlížeče Internet Explorer, který ani doposud nebyl schopen podporovat dnes již základní Javascriptové konstrukce.

4. Axios

   Knihovna pro Javascript, která nám umožňuje vytvářet asynchronní požadavky založené na promisech[33]. Používá se pro komunikaci mezi backendem a klientským prohlížečem.

5. Chart.js

   Knihovna na tvorbu grafů v Javascriptu[34], která podporuje velké množství a typů grafů. Aplikace vyvíjená v rámci této diplomové práce využívá její sloupcové a spojnicové grafy.

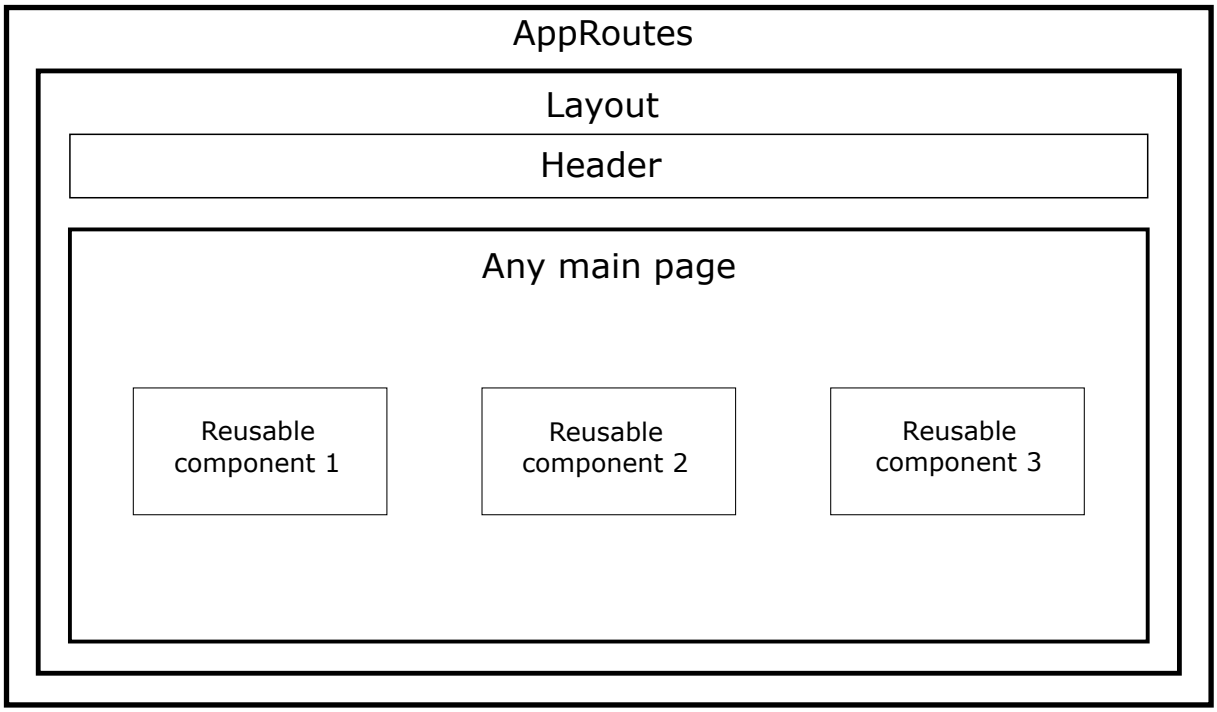
6. React-Boobstrap

   Klasický bootstrap framework, který byl přepsán do Reactu. Každá vlastnost je zde reprezentována React komponentou, které je potřeba nejdříve naimportovat. Je využívána napříč celou aplikací od jednoduchých tlačítek po složitější komponenty, jako jsou například modální okna.

5.4 Návrh frontend části

Díky tomu, že se jedná o již zmíněnou *Single Page Application*, tak je v celém projektu pouze jeden HTML soubor, a to *index.html*. Na tento soubor je napojena *root* React komponenta *AppRoutes*. Primárním cílem této komponenty je nadefinovat základní chování aplikace. Zaprvé se zde namapuje seznam URL cest na jednotlivé hlavní React komponenty. To znamená, že pokud máme například URL cestu */pivotal*, tak se na ni namapuje React komponenta *PivotalLayout*. Vždy když tedy navštívíme URL */pivotal*, zobrazí se nám *PivotalLayout* komponenta. Dále se zde vytváří *Redux store*, který bude popsán v kapitole 5.4.1 a načítají se zde potřebná data z externích zdrojů. Mezi tato data patří konfigurace projektu, veškeré jeho sprinty a pak také informace o použitých integracích, jako je například *Pivotal Tracker*.

Další komponentou je komponenta *Layout*, která zaobaluje celý viditelný obsah a skládá se ze dvou komponent. První komponentou je komponenta *Header*, která je neustále viditelná a slouží k navigaci v uživatelském rozhraní, a také obsahuje funkcionalitu pro změnu projektu a pro konfiguraci aktuálně zvoleného projektu. Druhou komponentou je placeholder pro jakoukoli jinou hlavní komponentu. Mezi tyto komponenty patří ty, které jsou namapovány na jednotlivé URL cesty a tvoří tak základní kameny uživatelského rozhraní. Tyto komponenty jsou nadefinovány v adresáři *pages*. Každá z těchto komponent se pak skládá z menších, znovupoužitelných komponent, které jsou umístěny v adresáři *components*, respektive *components/utils*. Tuto strukturu znázorňuje obrázek číslo 6.

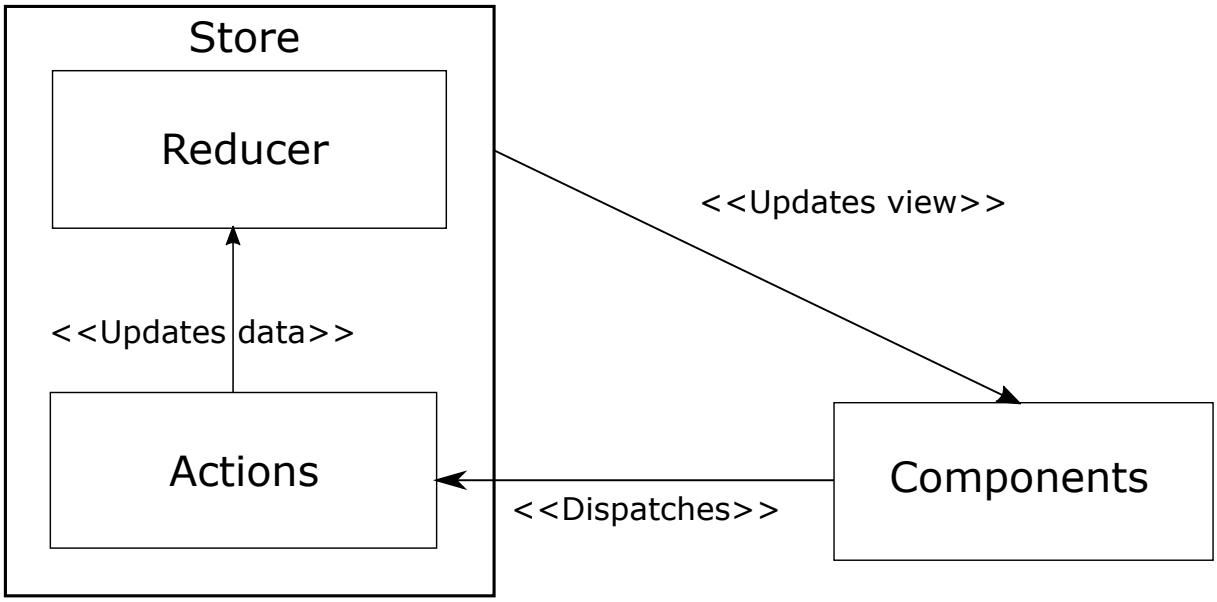


Obrázek 6: Struktura komponent

Dále se zdrojové kódy frontendu skládají z adresáře *axios-requests* a *mappers*. Pod *axios-requests* si lze představit obdobu vrstvy přístupu k datům z externích zdrojů zmíněnou v kapitole 5.2.5. Hlavním rozdílem je, že jsou tyto požadavky vykonávány asynchronně pomocí *Promise*. Tato technika je podrobněji popsána v kapitole 7.3. *Mappers* soubory jsou pak obdobou backend vrstvy mapperů z kapitoly 5.2.6.

5.4.1 Redux

Aplikace také obsahuje adresáře *reducers* a *actions*. Tyto adresáře obsahují kód potřebný pro použití *Redux* knihovny[18]. Reducer si můžeme představit jako předdefinovaný objekt, který uchovává veškerá data části aplikace a také operace, jak s těmito daty pracovat. Pro představu například *ProjectReducer* obsahuje veškeré informace o daných projektech, jako je jejich konfigurace nebo sprinty a obsahuje operace pro práci s těmito daty, jako je například *CHANGE_EXPECTED_NUMBER_OF_SPRINTS*. Tyto operace mají určité požadavky a omezení a její konkrétní příklad bude dále popsán v kapitole 7.4. Ke každému reduceru pak existuje soubor v adresáři *actions*. Ke zmíněnému *ProjectReducer* tedy existuje soubor *project-actions.js*, který obsahuje operace pro změny dat nadefinováne v reduceru. Operace z tohoto adresáře lze naimportovat do jiných komponent a tím pak změny dat vyvolat. Kombinace reducerů a akcí se pak nazývá *store*, který je potřeba napojit na každou komponentu, která s daty chce manipulovat. Tento proces zobrazuje obrázek číslo 7.



Obrázek 7: Redux diagram

Jak na obrázku vidíme, komponenty *dispatchují akce*, které odchytí reducer a aktualizuje data. Tato nová data jsou následně promítnuta zpět na komponenty, které se podle potřeby překreslí.

6 Lokální spuštění aplikace

Tato kapitola podrobně popisuje, jaké kroky je nutno provést k tomu, aby byla aplikace úspěšně spustitelná v lokálního prostředí. Před spuštěním aplikace je nutné naninstalovat následující nástroje:

1. Java 8 JDK
2. Maven
3. NodeJS + NPM (doporučená verze 6.10.1)
4. Mongo DB Community Server

Také je nutno zvlášť spustit backend, jehož součástí je databáze a sestavit frontend. Před spuštěním backendu je dále nutno spustit databázi Mongo DB. Toho docílíme spuštěním příkazu *mongod* v příkazové řádce.

6.1 Sestavení backendu

V dalším kroku je nainstalování Maven projekt *projectrisks-api*. V jeho kořenovém adresáři tedy spustíme příkaz *mvn clean install*. Nyní zbývá pouze sestavit samotnou aplikaci v Maven projektu *projectrisks-core*. V jeho kořenovém adresáři je opět potřeba spustit příkaz *mvn clean install*. Tyto kroky je nutno vykonat ve správném pořadí, jelikož aplikaci nelze spustit bez běžící databáze a také proto, protože projekt *projectrisks-core* má projekt *projectrisks-api* jako závislost. Spuštění tohoto skriptu také provede všechny unit testy, kterých je několik desítek a týkají se zejména mapování dat.

Při prvním spuštění bude samozřejmě databáze prázdná. Aby nebylo nutno celou databázi plnit, je v adresáři *projectrisks-core/src/main/frontend/scripts/temp* soubor *projects.json*, který obsahuje předpřipravená data. Tato data byla vygenerována skriptem *copyToLocal.sh*, který provádí export databáze a je jej možno naimportovat skriptem *importFromLocal.sh*. Oba tyto skripty se nacházejí v adresáři *projectrisks-core/src/main/frontend/scripts*.

Testovací data obsahují celkově dva záznamy v kolekci *projects*. Zaprvé se jedná o projekt *Main Project*, který obsahuje kompletní konfiguraci projektu, tedy předpokládaný počet sprintů, zákazníkem požadovaný poměr požadavků ku požadavkům nalinkovaných k testům a také zákazníkem požadovaný poměr požadavků ku požadavkům nalinkovaným k vývojovým úkolům. Dále obsahuje kompletní data pro pět sprintů, jejich metriky a také data pro potřeby integrace k externím nástrojům. Tyto integrace slouží pouze pro testovací účely a nevedou k žádným skutečným projektům.

Druhým záznamem je pak projekt s názvem *Real Project*, který obsahuje odkaz na skutečný *Pivotal Tracker* projekt.

6.2 Sestavení frontendu

Sestavení frontendu je trochu složitější. Zaprvé je nutno v adresáři *projectrisks-core/src/main/frontend* spustit příkaz *npm install*, kterým stáhneme všechny externí zavislosti, které jsou nadefinovány v souboru *package.json*. V tomto souboru je také závislost na Webpack2, který se tímto příkazem nainstaluje také, ale je jej potřeba nainstalovat také globálně. Globální instalace je potřeba z toho důvodu, aby byl příkaz *webpack* dostupný v příkazové řádce, kde je potřeba, aby mohl být spouštěn pomocí skriptů. Této globální instalace docílíme spuštění příkazu *npm install webpack@3.8.1 –global*.

Nyní je potřeba spustit skript *build_client.sh* v adresáři *projectrisks-core/src/main/frontend*. Tento skript pomocí Webpacku a Babelu3 sestaví ze všech javascriptových souborů soubor jeden, který je zároveň díky Babelu převeden na verzi Javascriptu, které rozumí internetové prohlížeče. Tento vytvořený soubor, který se jmenuje *bundle.js* je vložen do backend adresáře */resources/static/*, odkuď jej dodává backend při požadavku na *intex.html*. V závislosti na výkonu počítače může první spuštění tohoto skriptu trvat značnou dobu.[1] Tento skript pak běží na pozadí, sleduje změny ve zdrojových kódech a v případě, že byl nějaký soubor změněn, je výsledný soubor znovu sestaven. V souvislosti s tímto procesem je také pomocí *source maps*[36] vytvořen soubor *bundle.js.map*, který slouží k tomu, abychom byli schopni ve vygenerovaném souboru najít původní umístění zdrojových kódů.

6.3 Samotné spuštění aplikace

Po nainstalování a sestavení všech potřebných prerekvizit již zbývá pouze spustit samotnou aplikaci pomocí skriptu *runserver.sh*, popřípadě *runserverdebug.sh* pro debugování. Oba tyto skripty se nacházejí v kořenovém adresáři projektu *projectrisks-core*. V případě, že se rozhodneme aplikaci debugovat, je také potřeba nakonfigurovat vzdálený debugger z vývojového prostředí na port 5005.

Nezávisle na typu spuštění by měl skript při úspěšném spuštění vypsat zprávu *Application running!*. V tomto případě pak aplikace běží na portu 8090 a v prohlížeči je dostupná na adrese *http://localhost:8090/*.

[1] Pro představu na jednom ze zařízení, na kterém byla aplikace vyvíjena - Ultrabook s procesorem Intel i3, bez SSD mohlo prvotní spuštění trvat i několik minut.

7 Implementační detaily vybraných částí aplikace

V této kapitole budou popsány vybrané implementační detaily a problémy, které se při vývoji aplikace vyskytly. Dále se tato kapitola zabývá použití některých technologií, které lze považovat za stavební kameny asynchronních webových aplikací psaných v Javascriptu.

7.1 Spring repository + MongoDB

Jak bylo řečeno v kapitole 5.2.3, aplikace používá abstrakci *Spring Data Repository*, která dramaticky snižuje množství kódu, které je potřeba pro přístup do datové vrstvy. Pro tyto potřeby se využívá rozhraní *ProjectRepositorySpring*, které rozšiřuje rozhraní *MongoRepository*.

Díky tomu není vůbec potřeba implementovat základní a nejpoužívanější funkcionalitu a operace jako *save()*, *update()* nebo *findAll()* jsou již naimplementovány v nadřazené třídě.

Další obrovskou výhodou je to, že je tato abstrakce schopna sama vygenerovat vlastní databázové dotazy pouze ze signatury metody. Metodu v rozhraní se signaturou *Project findByProjectName(String projectName)* pak není nutno implementovat a engine je sám schopen rozpoznat, že se hledá projekt podle jeho jména.

Vše, co bylo potřeba pro potřeby přístupu k datům pak znázorňuje následující výpis zdrojového kódu.

```
1 package projectrisks.repository;
2
3 import org.springframework.data.mongodb.repository.MongoRepository;
4 import projectrisks.domain.project.Project;
5
6 public interface ProjectRepositorySpring
7    extends MongoRepository<Project, String> {
8     Project findByProjectName(String projectName);
9 }
```

Výpis 2: ProjectRepositorySpring.java

7.2 Integrace na externí zdroje

Aplikace se integruje na dvě externí API.

1. Pivotal Tracker integrace

   Integrace na API tohoto systému bylo poměrně jednoduché. Pro autentikaci se zde používá HTTP header *X-TrackerToken*, jehož hodnotu lze získat na stránkách projektu. Pro získání iterací se volá klasický GET požadavek vyvolaný Spring třídou *RestTemplate*, kterému se pouze nastaví potřebné headery.

   Velmi příjemná byla také práce s *pagingem*[2], které toto API podporuje. Jediné, co bylo potřeba udělat, bylo nastavit URL parametr *offset* na požadovanou hodnotu. Implementační detaily této funkcionality jsou k dispozici v kapitole 7.7.

2. Version One integrace

   Integrace na API nástroje Version One byla o něco málo komplikovanější, jelikož je třeba použít jejich třídy a connector. Tyto třídy lze do aplikace vložit pomocí maven artefaktu *VersionOne.SDK.Java.APIClient*. Používání tohoto API se dá velmi dobře přizpůsobit, nicméně pro potřeby vyvíjené aplikace to způsobilo v kombinaci s ne úplně přehlednou dokumentací pomalejší vývoj a hloubkové zkoumání dokumentace.

7.3 Promise

Promise je Javascriptový objekt, který slouží k vykonávání asynchronních operací a reprezentuje její stav a její výslednou hodnotu. Jedná se o jistou náhradu za *callbacky*, u kterých při vnoření jednotlivých asynchronních operací docházelo k nepřehlednému a těžko udržovatelného kódu, který bývá označován jako *callback hell*[37].

Jakmile je asynchronní operace spuštěna, má její promise objekt stav *pending*. Asynchronní metoda, která tento promise vrací a která je v tomto stavu nikdy neskončí. Pokud je promise úspěšně vykonána (například server vrátil odpověď s HTTP statusem 200), získá promise stav *fulfilled* a asynchronní metoda úspěšně vrátí data. V případě chyby pak promise objekt nabyde stavu *rejected* a metoda opět vrátí data - například chybovou hlášku, HTTP status apod. Praktický rozdíl mezi těmito stavy je zejména ten, že pokud je promise ve stavu rejected, víme, že na ni máme zareagovat jinak, než kdyby byla úspěšně vykonána. Například můžeme zobrazit chybové hlášení nebo nevolat další asynchronní funkci, která je závislá na úspěšném stavu první funkce.

Promisy lze tedy také řetězit, kdy data načtená jednou promise vstupujou jako vstupní hodnoty do promise následující. Tento případ užití popisuje následující příklad, který je použit v aplikaci.

[2]Paging je technika, díky které se nenačítají veškerá data daného zdroje, ale pouze například prvních 20 záznamů. V případě potřeby si pak lze vyžádat záznamy další.

Pokaždé, kdy uživatel navštíví stránky aplikace, dojde k asynchronnímu volání backendu na získání všech projektů. Jestliže je promise, která toto volání zprostředkovává úspěšně dokončena, tak dojde k dalšímu asynchronnímu volání backendu. Konkrétně se jedná o načtení dodatečných projektových dat z externích zdrojů a poté dojde k dočtení dat z nástroje Pivotal Tracker a Version One. Toto volání nemůže být iniciováno bez znalosti dat o projektu, jelikož tato data obsahují jména aplikací, které zde jsou zaregistrovány a také přístupové tokeny.

Konkrétní příklad této implementace popisuje následující, zjednodušený výňatek ze zdrojového kódu.

```
1  loadProjectDataRequest()//promise pending
2  .then(projectsData => {
3     //promise fulfilled
4     loadPivotalDataRequest(projectsData)
5     .then(pivotalData => {});
6
7     loadVersionOneDataRequest(projectsData)
8     .then(versionOneData => {});
9  })
10 .catch(error => {
11    //promise rejected, handle error
12    console.log(error)
13 });
```

Výpis 3: Příklad promise

7.4 Reducer operace

Pokud chceme v aplikaci změnit stav operace, musíme použít Redux 5.4.1. Redux obsahuje reducery, které se skládají z dat a operací, které modifikují data.

Data mohou být velmi jednoduchý objekt, jako je například následující výpis kódu.

```
1  const initialVersionOneState = {
2      iterations: {}
3  }
```

Výpis 4: Reducer data

Operace pak reprezentuje jedna funkce, která přijímá dva argumenty. První argument je aktuální stav aplikace a druhý je akce, která chce tento stav měnit. Na typu této akce se pak

rozhodne, která konkrétní operace se vykoná. Příklad této funkce lze vidět na následujícím výpisu kódu.

```
const VersionOneReducer = (state = initialVersionOneState, action) => {
    switch(action.type) {
        case 'LOAD_VERSION_ONE_DATA':
            return Object.assign({}, state, {
                data: action.payload
            });

        case 'CHANGE_ACTIVE_CHART':
            return Object.assign({}, state, {
                activeChart: action.payload
            });

        default:
            return state;
    }
}
```

Výpis 5: Reducer funkce

Každý jednotlivý příkaz *case* reprezentuje jeden typ akce, kterou lze s daty vykonat. Akce kromě svého typu nese ještě *payload*, což jsou nová data. Je nutné si dát pozor, aby žádná akce nemodifikovala stávající stav, ale vracela vždy kompletně nový objekt. O toto se v našem případě stará metoda *Object.assign({}, state, newData)*, kde první parametr značí, že se má vytvořit úplně nový objekt.

7.5 Uživatelsky vytvořené metriky

Krása této funkcionality spočívá v tom, že je kompletně celá generická a dynamicky vytvořena. Žádná data, se kterými se zde pracuje nejsou v aplikaci napevno zakódována, ale dočítají se z databáze. Při vstupu na stránku *Custom metrics* se aplikace podívá do databáze a dotáhne všechny typy metrik, které jsou pro daný projekt k dispozici, to znamená ty metriky, pro které jsou v databázi příslušná data. Po výběru konkrétní metriky se také načtou její jednotlivé typy dat, což například pro metriku *Budget Data* znamená hodnoty *Planned Costs* a *Actual Costs*.

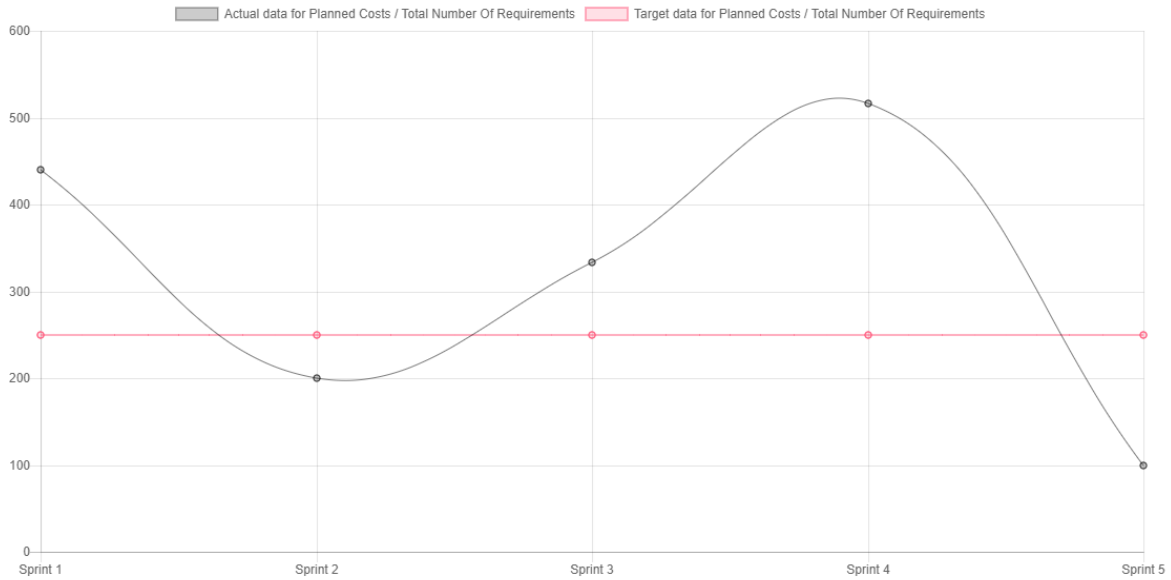
Dále je nutno zvolit výraz, podle kterého se bude metrika vyhodnocovat. Vstupní pole pro tyto potřeby povoluje zadat znak x reprezentující data první metriky a znak y, který reprezentuje data metriky druhé. Další povolené znaky jsou *+,-,*,/,(a)*. Tento výraz se vyhodnocuje pomocí

Javascriptové metody *eval()*, která je po nahrazení znaků x a y skutečnými hodnotami schopna jakýkoli výraz vyhodnotit[3].

Posledním vstupem pro tuto funkcionalitu je cílová hodnota, kterou je nutno překonat, aby byl výsledek metriky požadován za úspěšný. Tato hodnota může být například konstanta.

Výsledek těchto vstupů pak zobrazuje obrázek číslo 8. Vstupní data pro tento graf byly metriky *Budget Data* a *Velocity Data*. Konkrétní data pak byla *Planned Costs* a *Total Number Of Requirements*.

Výraz byl zvolen jako x / y a cílová hodnota byla zvolena jako konstanta *250*.



Obrázek 8: Ukázkový graf s uživatelsky vytvořenou metrikou

Metriku tedy můžeme volně interpretovat tak, že na každý jednotlivý požadavek sprintu musí být vyčleněno alespoň 250 jednotek prostředků. Z grafu můžeme vyčíst, že cílová hodnota (červená barva) byla dosažena ve sprintech 1, 3 a 4, ale ve sprintech 2 a 5 nikoliv.

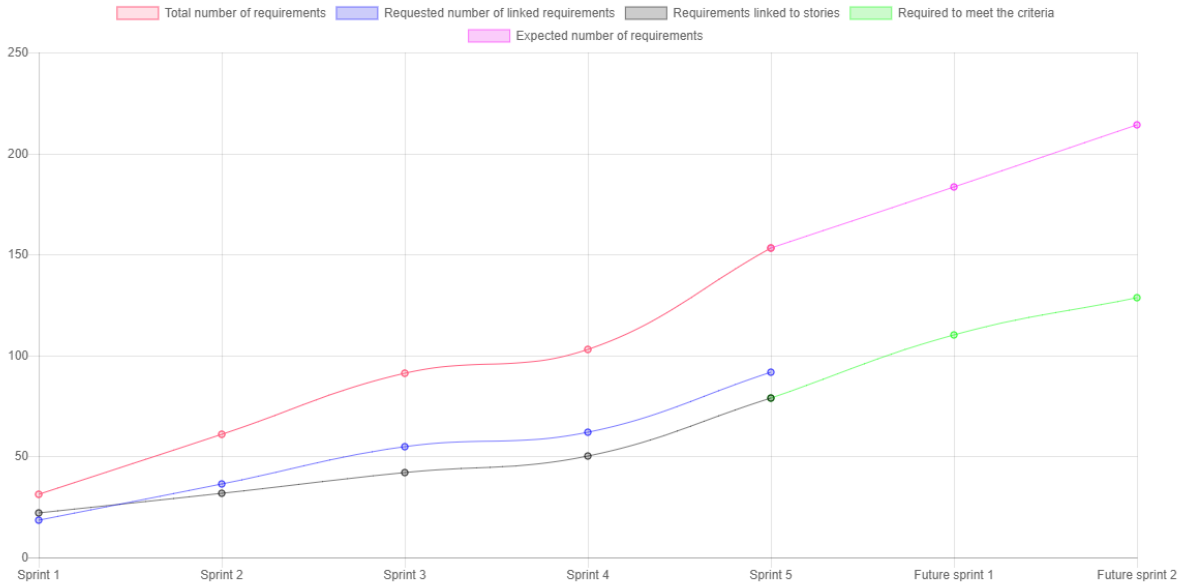
Výsledná data této funkcionality lze také vyexportovat do Excelu pomocí *npm* knihovny *react-excel-workbook*[38], která představuje nejjednodušší práci s exportem dat do Excelu, se kterou jsem se kdy setkal a mohu pro práci s React aplikacemi silně doporučit.

[3]Je vhodné zmínit, že metoda *eval()* je potencionálně nebezpečná pro vyhodnocování uživatelských vstupů, jelikož může vést ke spouštění škodlivých kódů na stránce. Příklad v této aplikaci však omezuje uživatelský vstup na velmi omezený počet znaků, čímž se tato rizika eliminují.

7.6 Dopočet hodnot budoucích sprintů

V případě, že daný projekt v aplikaci má nakonfigurovanou hodnotu předpokládaného počtu sprintů, a zároveň pro všechny z tohoto počtu neexistuje záznam v databázi, aplikace u zákaznických metrik dopočte, jakých dat je v budoucnu potřeba dosáhnout, aby byly zákaznické požadavky splněny.

Dopočet hodnot budoucím sprintům je jeden z aspektů této aplikace, který by se dal vylepšit inteligentnějším algoritmem, jelikož má velký potenciál využitelnosti, ale jeho současná implementace používá pouze jednoduchých matematických operací, zejména pak aritmetického průměru. Obrázek číslo 9 reprezentuje snímek z aplikace, který znázorňuje tento dopočet pro metriku nalinkování požadavků k programátorským úkolům.



Obrázek 9: Ukázkový graf s dopočtenými hodnotami

Graf zobrazuje několik datových sad. Červená barva znázorňuje celkový počet všech požadavků. Šedá barva znázorňuje celkový počet požadavků nalikovaných na programátorské úkoly a modrá barva reprezentuje minimum, kterého je potřeba dosáhnout, aby byly splňeny zákaznické požadavky. Tento požadavek je pro tento graf stanoven na 60 procent, což znamená, že 60 procent z celkového počtu požadavků musí být nalinkovaných. Z grafu lze vyčíst, že tento požadavek byl splňen pouze v prvním sprintu, v dalších již nebylo dostatek úkolů nalinkováno.

Graf také zobrazuje další dvě datové sady. Fialovou barvou je reprezentován předpokládaný budoucí počet sprintů a zelená barva reprezentuje budoucí nejmenší nutný počet nalinkovaných požadavků, aby byl splněn zákaznický požadavek. Obě hodnoty byly vypočteny zprůměrováním existujících dat.

7.7 Pivotal Tracker Paging

Pivotal Tracker rozhraní podporuje velmi užitečný mechanismus pro *pagination*, který můžeme regulovat pomocí URL parametrů *offset* a *limit*. Vyvíjená aplikace používá výchozí hodnotu pro *limit* a pracuje tedy pouze s parametrem *offset*. Následující řádek popisuje, jak může vypadat ukázkový požadavek využívající *pagination*.

GET projects/1/stories?offset=25&limit=300&envelope=true[39]

Tento požadavek pak vygeneruje následující odpověď:

```
{
   "http_status": "200",
   "project_version": 42,
   "pagination":
   {
       "total": 1543,
       "limit": 300,
       "offset": 25,
       "returned": 298
   }
}
```

Na tuto odpověď pak reaguje frontend, který umožňuje uživateli načíst starší data v případě, kdy se proměnná *pagination.offset* nerovná nule (nemáme již nejstarší data) a nebo načíst novější data v případě, kdy není proměnná *pagination.offset* rovna hodnotě *pagination.total - 1* (nemáme již nejnovější data).

8 Závěr

Cílem této práce bylo čtenáři podat ucelené informace o základních rizikách projektu, které lze aplikovat na jakékoli softwarové dílo většího rozsahu. Tato rizika byla nejdříve obecně popsána a poté byla zkoumána hlouběji v závislosti na jednotlivých vývojových fázích.

Hlavní část diplomové práce se pak týkala praktického návrhu a implementace vlastního nástroje pro řešení, analýzu a prevenci rizik projektu. V rámci tohoto úkolu vznikla webová aplikace, která používá dokumentovou databázi Mongo DB, Spring Boot Java backend a javascriptový frontend implementovaný v Reactu JS.

Primární funkcionalitou této aplikace je zobrazovat v grafech určité typy dat, ze kterých pak může projektový manažer, respektive jakýkoli jiný odpovědný člověk usoudit, že se projekt nevyvíjí korektním směrem. Ke klasickému zobrazení existujících dat existují také funkcionality, které by tomuto procesu měly napomoci. Jedná se například o predikci hodnot v budoucích vývojových iteracích, a nebo nadefinování požadavků, které musí daná metrika splňovat.

V otázce dalšího vývoje lze aplikaci rozšiřovat jak ve funkčních, tak i nefunkčních požadavcích. Mezi kandidáty na vylepšení funkčních požadavků stojí za zmínku vylepšení integrace na externí nástroje, jako je například Pivotal Tracker, nebo implementace více inteligentní predikce dat pro budoucí iterace.

Z pohledu vlastního zhodnocení smysluplnosti této diplomové práce a samotného oboru týkajícího se analýzou rizik projektu jsem dospěl k závěru, že by se této problematice mělo věnovat mnohem více času, usíli a zdrojů, jelikož velká část softwarových produktů končí neúspěchem. Pro představu, článek z roku 2016[40] citující data z *IDC (International Data Corpotation)* informuje o tom, že až 25 procent IT projektů představují neúspěch a dalších až 20 procent nevykazuje návratnost investice. Z těchto dat a vzhledem ke stále se zvyšující náročnosti IT projektů a s rostoucí digitalizaci světa všude kolem nás považuji za zřejmé, že tato disciplína bude mít v IT nezastupitelnou roli.

Literatura

[1] Top Ten Lists of Software Project Risks:Evidence from the Literature Survey: http://www.iaeng.org/publication/IMECS2011/IMECS2011_pp732-737.pdf

[2] RISK FACTORS IN SOFTWARE DEVELOPMENT PHASES: http://eujournal.org/index.php/esj/article/viewFile/2624/2485

[3] Overengineering https://en.wikipedia.org/wiki/Overengineering

[4] KISS principle https://en.wikipedia.org/wiki/KISS_principle

[5] Don‘t repeat yourself https://en.wikipedia.org/wiki/Don%27t_repeat_yourself

[6] A Review of Risk Management in Different Software Development Methodologies http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.685.6705&rep=rep1&type=pdf

[7] Waterfall model https://en.wikipedia.org/wiki/Waterfall_model

[8] Bus factor https://en.wikipedia.org/wiki/Bus_factor

[9] Five agile metrics you won't hate https://www.atlassian.com/agile/project-management/metrics

[10] 7 of the Top Agile Project Management Software - Capterra Blog http://blog.capterra.com/agile-project-management-software

[11] Risk Management for Jira https://marketplace.atlassian.com/plugins/com.ja.jira.plugin.report.riskman

[12] Risk matrix https://en.wikipedia.org/wiki/Risk_matrix

[13] 5 Handy Uses for Issues in VersionOne https://blog.versionone.com/5-handy-uses-for-issues-in-versionone/

[14] ISO/IEC 15504 https://en.wikipedia.org/wiki/ISO/IEC_15504

[15] Convention over configuration https://en.wikipedia.org/wiki/Convention_over_configuration

[16] Spring Framework https://en.wikipedia.org/wiki/Spring_Framework#Spring_Boot

[17] React (JavaScript library) https://en.wikipedia.org/wiki/React_(JavaScript_library)

[18] Redux https://github.com/reactjs/redux

[19] NoSQL https://en.wikipedia.org/wiki/NoSQL

[20] MongoDB https://en.wikipedia.org/wiki/MongoDB

[21] Morphium - Java Object Mapper and Caching Layer for MongoDB https://github.com/sboesebeck/morphium

[22] Spring Data MongoDB https://projects.spring.io/spring-data-mongodb/

[23] Spring Data Commons https://docs.spring.io/spring-data/commons/docs/current/reference/html/

[24] ECMAScript https://en.wikipedia.org/wiki/ECMAScript

[25] Smarter Java development http://www.javaworld.com/article/2076468/core-java/smarter-java-development.html

[26] General data-binding package for Jackson (2.x): works on streaming API (core) implementation(s) https://github.com/FasterXML/jackson-databind

[27] Apache Log4j 2 https://logging.apache.org/log4j/2.x/

[28] Single-page application https://en.wikipedia.org/wiki/Single-page_application

[29] npm (software) https://en.wikipedia.org/wiki/Npm_(software)

[30] Node.js https://nodejs.org/en/

[31] Webpack https://webpack.github.io/

[32] Babel https://babeljs.io/

[33] Axios https://www.npmjs.com/package/axios

[34] Chart.js http://www.chartjs.org/

[35] React-Bootstrap https://react-bootstrap.github.io/components.html

[36] Introduction to JavaScript Source Maps - HTML5 Rocks https://www.html5rocks.com/en/tutorials/developertools/sourcemaps/

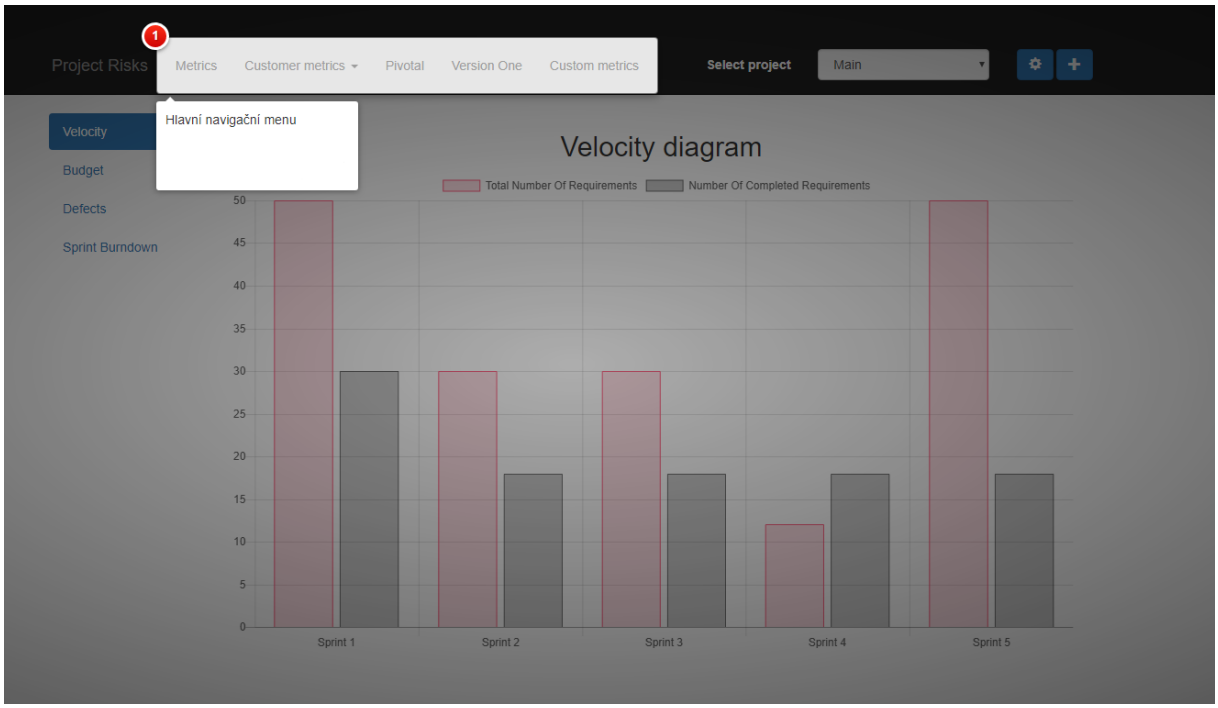
[37] Callback Hell http://callbackhell.com/

[38] react-excel-workbook https://www.npmjs.com/package/react-excel-workbook

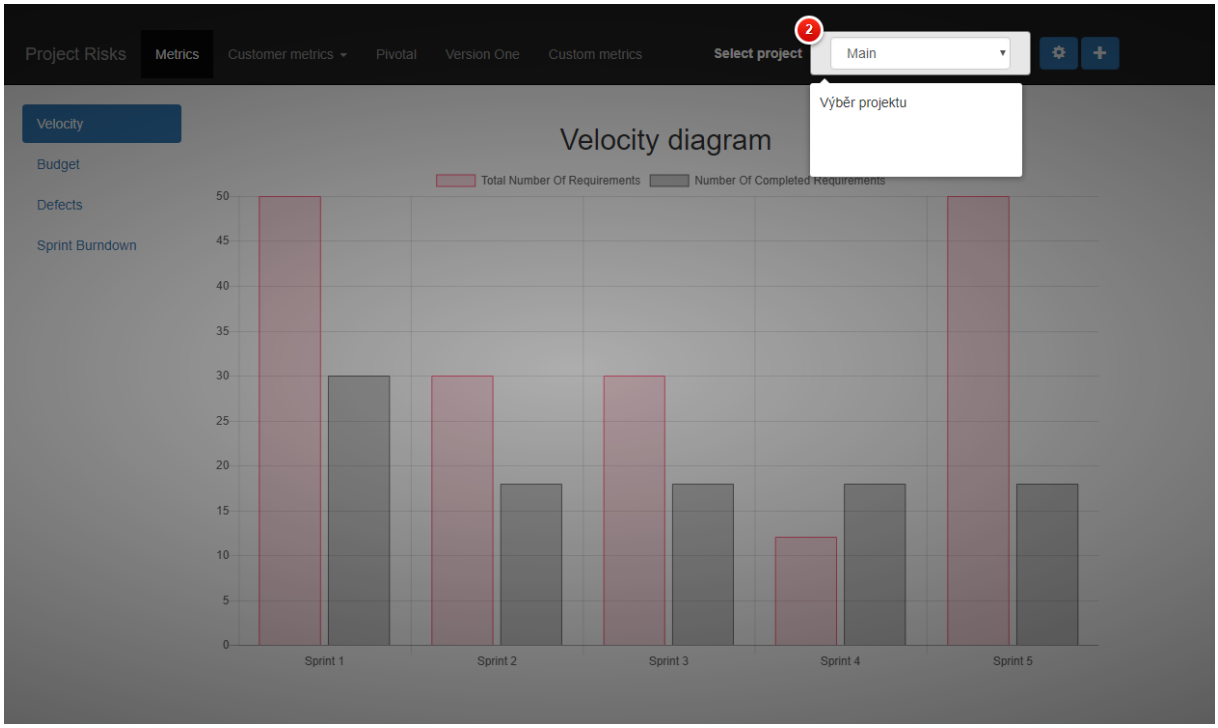
[39] Pivotal Tracker API https://www.pivotaltracker.com/help/api#Paginating_List_Responses

[40] Why Do Information Technology Projects Fail? https://www.sciencedirect.com/science/article/pii/S187

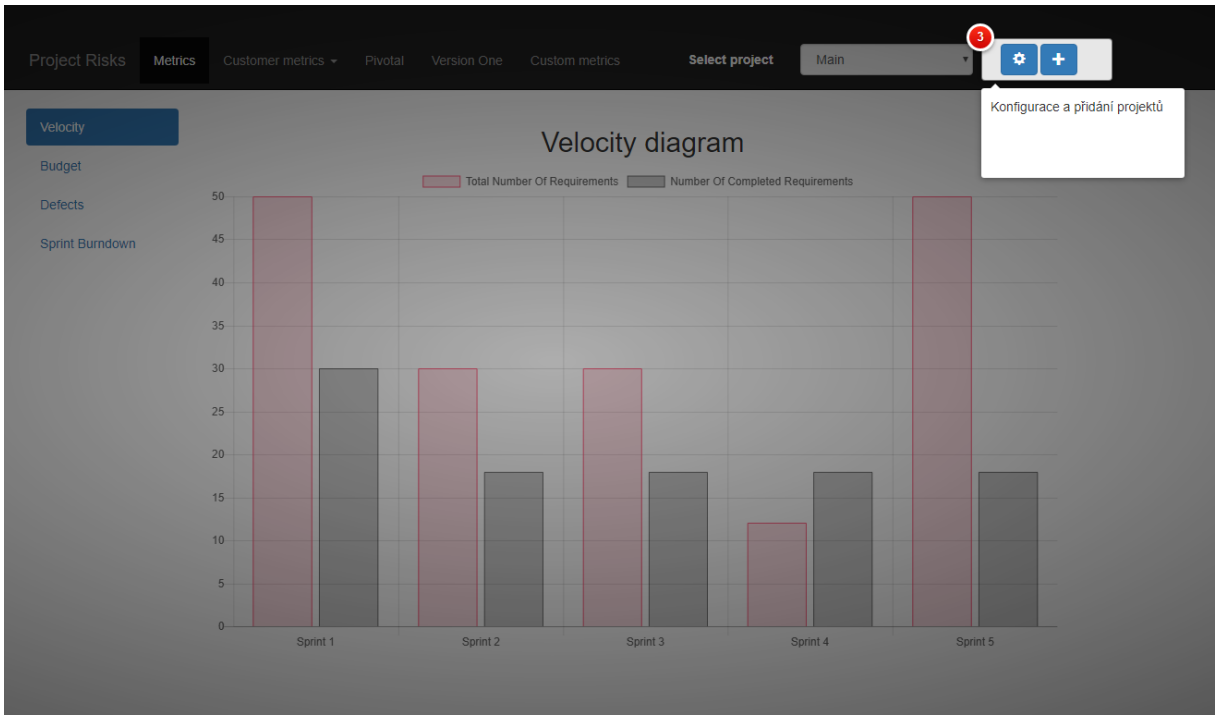
A Grafický popis uživatelského rozhraní - hlavní stránka



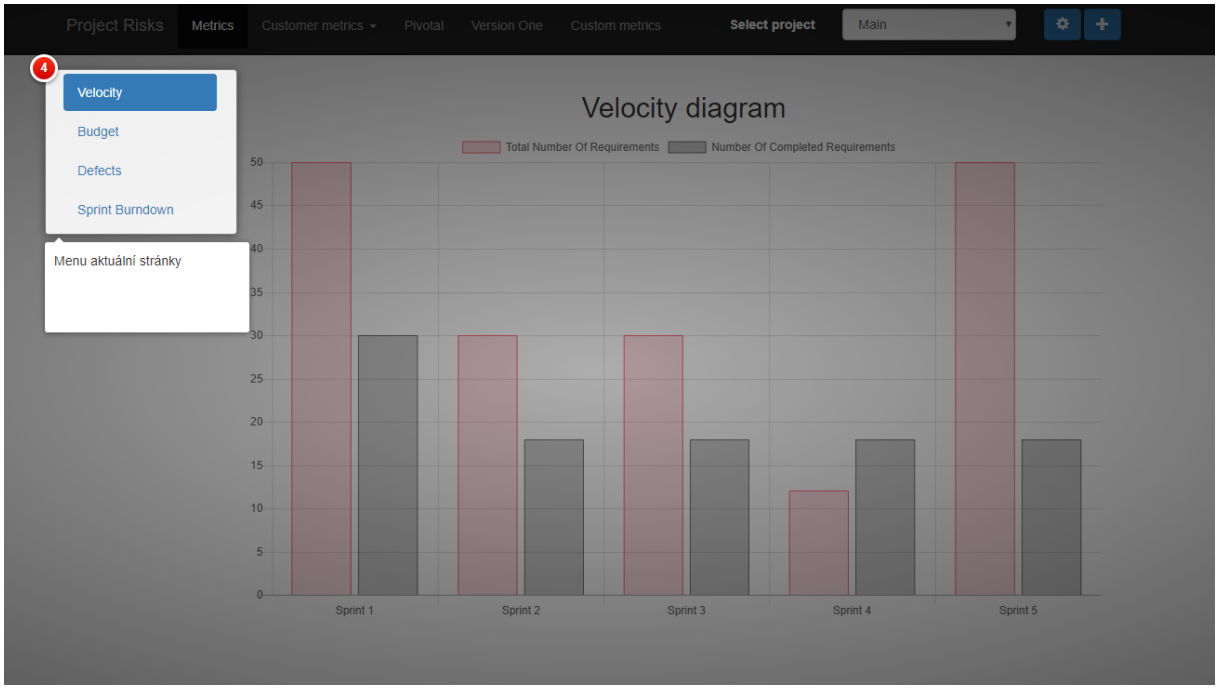
Obrázek 10: Hlavní menu



Obrázek 11: Výběr projektu



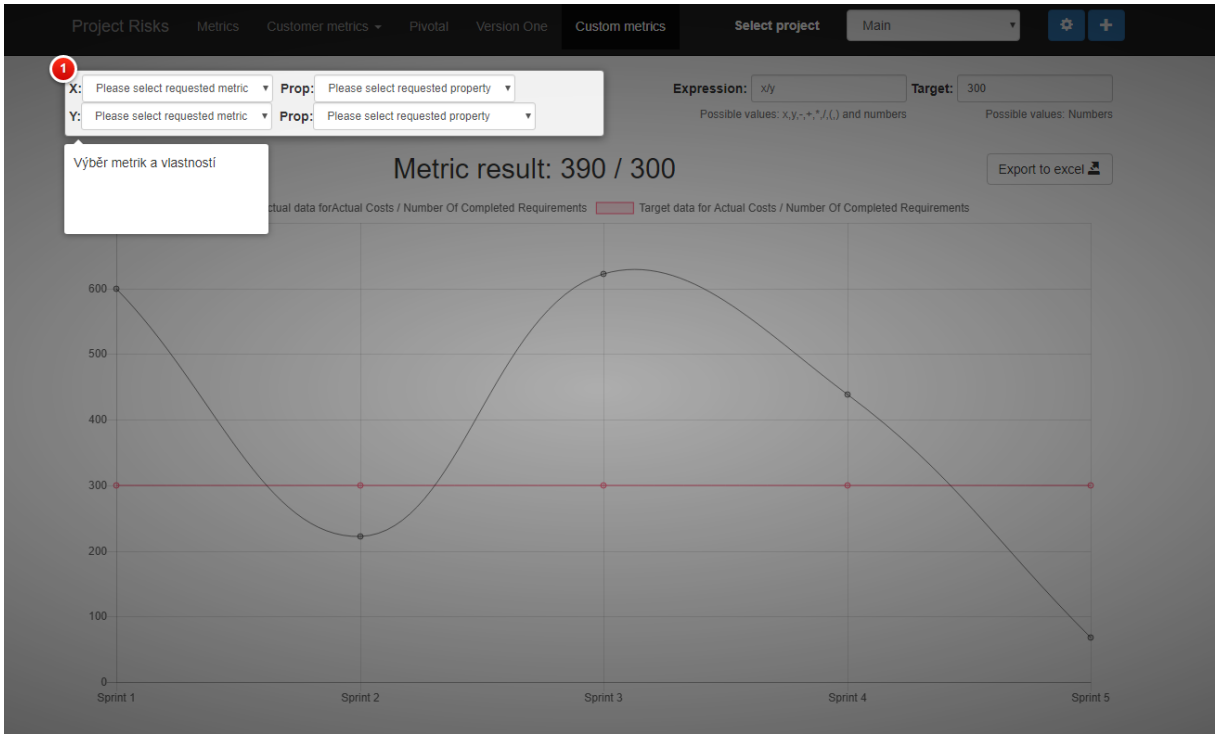
Obrázek 12: Konfigurace a přidání projektu



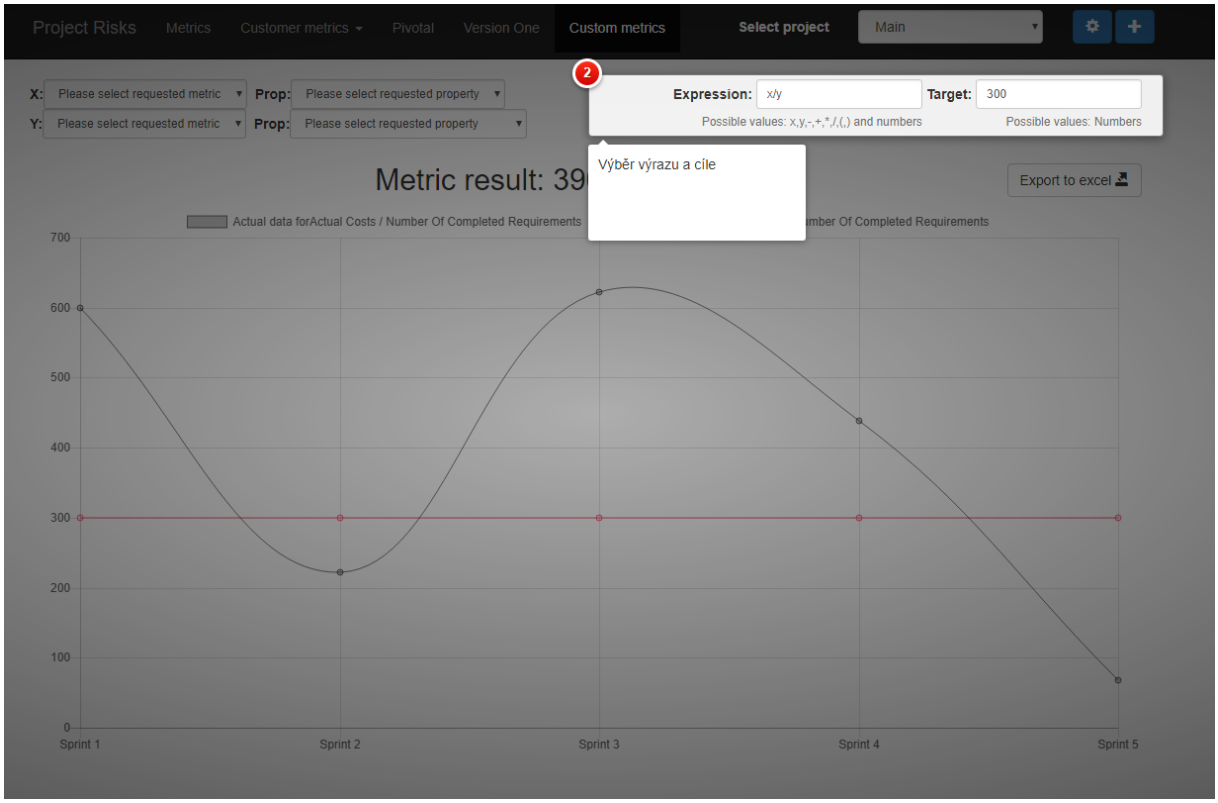
Obrázek 13: Menu aktuální stránky

Obrázek 14: Hlavní obsah stránky

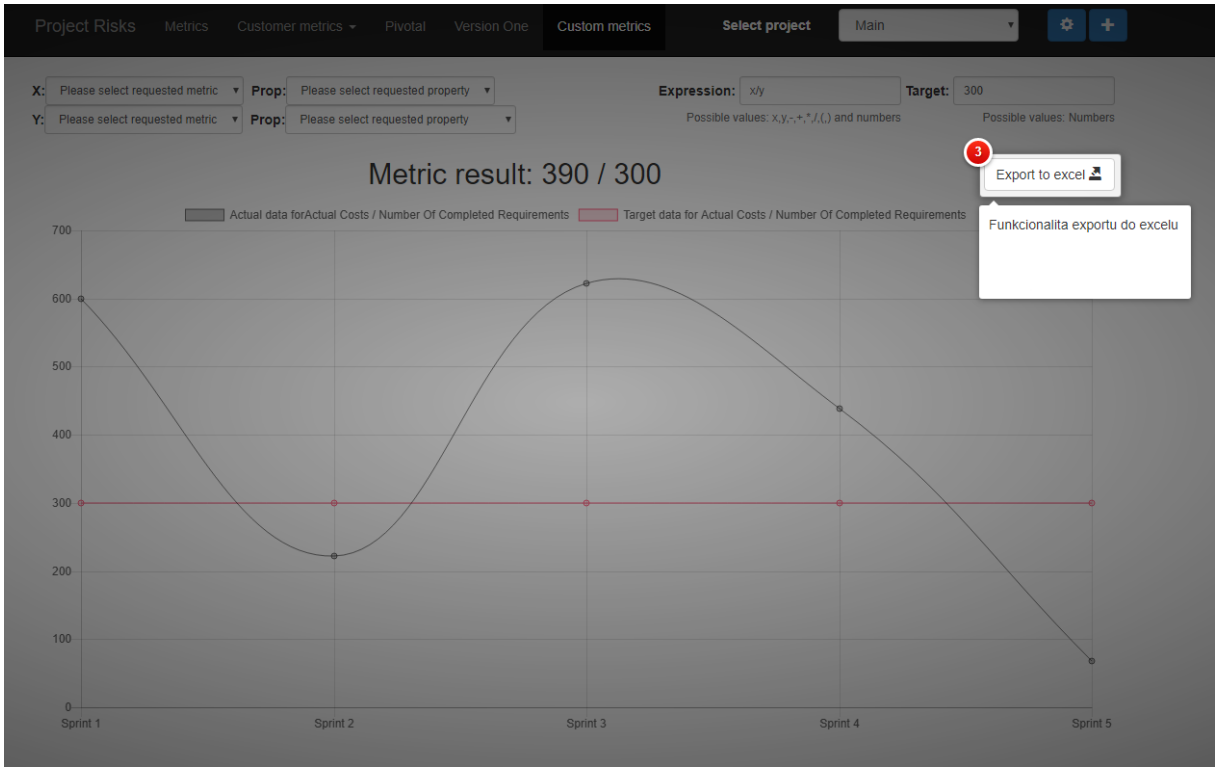
B Grafický popis uživatelského rozhraní - uživatelsky definované metriky



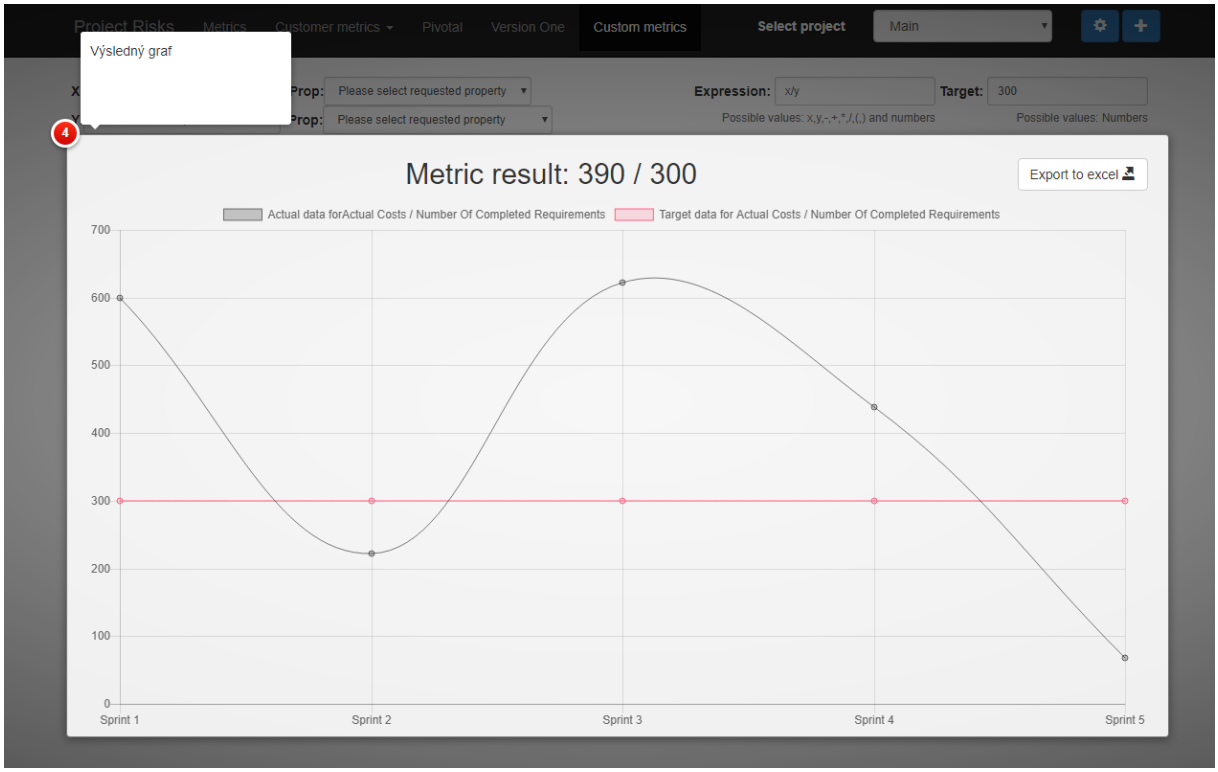
Obrázek 15: Výběr metrik a vlastností



Obrázek 16: Výběr výrazu a cíle



Obrázek 17: Export do excelu



Obrázek 18: Výsledný graf metriky